Datacenter Architectures for the Microservices Era

by

Seyedamirhossein Mirhosseininiri

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2021

Doctoral Committee:

Professor Thomas Wenisch, Chair
Professor David Blaauw
Professor Trevor Mudge
Professor Satish Narayanasamy
Professor Josep Torrellas, University of Illinois at Urbana-Champaign

Seyedamirhossein Mirhosseininiri

miramir@umich.edu

ORCID iD: 0000-0001-6501-6087

To the love of my life, Farnoosh.

ACKNOWLEDGEMENTS

This thesis is the result of many years of hard work that simply would not have been possible without the continuous support of many people. Undoubtedly, I could not have done this without the unwavering support and mentorship from my advisor Professor Thomas Wenisch. Tom gave me the freedom to pursue any research direction I was passionate about and allowed me to integrate multiple topics into my research. I would also like to particularly thank Professor Josep Torrellas, my original advisor at the University of Illinois at Urbana-Champaign, who taught me a lot of technical and research-related concepts and provided me with ample support, especially when I had to move from UIUC to Michigan. I would also like to thank all my industry collaborators and mentors, especially Geoff Blake from Amazon as well as Sameh Elnikety and Ricardo Bianchini from Microsoft Research who I learnt a lot of concepts from about datacenters and cloud computing. Finally, I would like to acknowledge my committee members, Professors David Blaauw, Trevor Mudge, and Satish Narayansamy for their help and support throughout my PhD.

The past six years have been extremely difficult for me, given all the pressure and stress induced by the Trump administration, and especially since I did not get to go home and see my family since the last time we were together at the airport in Tehran. I would like to thank my parents for their unconditional love and patience before and during this period, and for encouraging me to pursue my dreams, and making sure that I never had to worry about anything other than achieving my goals. Although this period was no easier for them than it was for me, they remained more than supportive all the time and gave me hope and encouragement every time I was about to give up.

I would also like to thank many friends who were like family to me all these years, including Pooyan Tirandazi, Hodjat Asghari Esfeden, Ali Rafei, Sajjad Najafi, Maryam Ahmadi, Amirhossein Herandi, Morteza Soltani, Mohsen Karimnejad, Morteza Taiebat, Mohsen Nabian, Brendan West, Harini Muthukrishnan, Akshitha Sriraman, Vaibhav Gogte, Ofir Weisse, and many others. Finally, I am forever grateful to my best friend—my brother—Antonio Franques who I was super lucky to have by my side all these years. I could not imagine this period without his friendship, help, and support in every possible way.

Last but not least, I would like you to thank my dearest Farnoosh, my fiance, to whom I dedicate this thesis. She has only recently joined my life but has made me happier than I could ever imagine, in a way like all of these difficult years never happened. I am grateful to her for the rest of my life and looking forward to a future full of love with her.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Modern internet services are shifting away from single-binary, monolithic services into numerous loosely-coupled microservices that interact via Remote Procedure Calls (RPCs), to improve programmability, reliability, manageability, and scalability of cloud services. Computer system designers are faced with many new challenges with microservice-based architectures, as individual RPCs/tasks are only a few microseconds in most microservices. In this dissertation, I seek to address the most notable challenges that arise due to the dissimilarities of the modern microservice-based and classic monolithic cloud services, and design novel server architectures and runtime systems that enable efficient execution of μs-scale microservices on modern hardware.

In the first part of my dissertation, I seek to address the problem of *Killer Microseconds*, which refers to μs-scale "holes" in CPU schedules caused by stalls to access fast I/O devices or brief idle times between requests in high throughput μs-scale microservices. Whereas modern computing platforms can efficiently hide ns-scale and ms-scale stalls through micro-architectural techniques and OS context switching, they lack efficient support to hide the latency of μs-scale stalls. In chapter II, I propose *Duplexity*, a heterogeneous server architecture that employs aggressive multithreading to hide the latency of killer microseconds, without sacrificing the Quality-of-Service (QoS) of latency-sensitive microservices. Duplexity is able to achieve 1.9$\times$ higher core utilization and 2.7$\times$ lower iso-throughput 99th-percentile tail latency over an SMT-based server design, on average.

In chapters III-IV, I comprehensively investigate the problem of tail latency in the context of microservices and address multiple aspects of it. First, in chapter III, I characterize the tail latency behavior of microservices and provide general guidelines for optimizing

computer systems from a queuing perspective to minimize tail latency. Queuing is a major contributor to end-to-end tail latency, wherein nominal tasks are enqueued behind rare, long ones, due to Head-of-Line (HoL) blocking. Next, in chapter IV, I introduce *Q-Zilla*, a scheduling framework to tackle tail latency from a queuing perspective, and CoreZilla, a microarchitectural instantiation of the framework. Q-Zilla is composed of the *Server-Queue Decoupled Size-Interval Task Assignment (SQD-SITA)* scheduling algorithm and the *Express-lane Simultaneous Multithreading (ESMT)* microarchitecture, which together seek to address HoL blocking by providing an "express-lane" for short tasks, protecting them from queuing behind rare, long ones. By combining the ESMT microarchitecture and the SQD-SITA scheduling algorithm, CoreZilla is able to improves tail latency over a conventional SMT core with 2, 4, and 8 contexts by 2.25×, 3.23×, and 4.38×, on average, respectively, and outperform a theoretical 32-core scale-up organization by 12%, on average, with 8 contexts.

Finally, in chapters V-VI, I investigate the tail latency problem of microservices from a cluster, rather than server-level, perspective. Whereas Service Level Objectives (SLOs) define end-to-end latency targets for the entire service to ensure user satisfaction, with microservice-based applications, it is unclear how to scale individual microservices when end-to-end SLOs are violated or underutilized. I introduce Parslo as an analytical framework for partial SLO allocation in virtualized cloud microservices. Parslo takes a microservice graph as an input and employs a Gradient Descent-based approach to allocate "partial SLOs" to different microservice nodes, enabling independent auto-scaling of individual microservices. Parslo achieves the optimal solution, minimizing the total cost for the entire service deployment, and is applicable to general microservice graphs.

In chapter VI, I study microservices that are shared across multiple end-to-end services, and must satisfy varying latency requirements for different request classes. I argue that sharing microservice instances across multiple services can reduce significantly the number of instances, especially for deployments with highly asymmetric latency constraints. I

propose a request scheduling mechanism, called μ*Steal*, which leverages preemptive work and resource stealing to schedule the arriving requests from multiple request classes to cores within an instance, seeking to maximize request throughput within an instance while ensuring all request classes meet their latency target. μSteal reduces the total number of instances required for several shared microservice deployments by 1.29$\times$ as compared to deploying multiple, segregated instance pools across request classes.

CHAPTER I

Introduction

Modern internet services are shifting away from single-binary, monolithic services into various loosely-coupled microservices, to enable rapid development, release, and frequent updates of cloud software [58, 59, 185, 183]. Microservice–based services are implemented as a Directed Acyclic Graph (DAG) composed of tens to hundreds of individual microservices, wherein each microservice node of the DAG is independently deployed and scaled. Microservice architectures have been adopted by major cloud-based companies, such as Facebook, Netflix, and Linkedin, as they significantly improve programmability, reliability, manageability, and scalability. For example, a Facebook news feed query may flow through a chain of microservices, such as Sigma (a spam filter), McRouter (a protocol router), Tao (a distributed social graph data store), and MyRocks (a user database) [183]. Computer system designers are faced with many new challenges with microservice-based architectures, as individual RPCs/tasks are only a few microseconds in most microservices [138, 9], exposing the end-to-end performance to many system-level overheads that used to be insignificant in conventional monlithic cloud architectures.

In this dissertation, I seek to address the most notable challenges that arise due to the dissimilarities of the modern microservice-based and classic monolithic cloud services, and design novel server architectures and runtime systems that enable efficient execution of μs-scale microservices on modern hardware. I particularly focus on μs-scale microservices

in Chapters III-IV and seek to improve their performance and efficiency. With μs-scale execution times of such microservice tasks, the I/O software stack's latency becomes comparable to computation time and must be aggressively optimized through hardware/software solutions. Furthermore, managing queuing delays and tail latency in case of short μs-scale tasks is much harder than classic ms-scale monolithic applications due to the overheads of task scheduling mechanisms—such as threading, synchronization, preemption, work steering, etc. In chapters V-VI, I investigate the tail latency problem of microservices from a cluster, rather than server-level, perspective. I first investigate how "partial" Service-Level Objectives (SLOs) or latency requirements must be imposed on individual microservices—given an end-to-end latency SLO—so each microservice may be scaled independently of the others. Finally, I study mixed-criticality microservices, which need to satisfy varying latency SLOs for multiple request classes originated from different end-to-end services, and investigate how requests can be scheduled in such environments to maximize throughput and efficiency while ensuring all request classes meet their SLO.

1.1 Killer Microseconds

We are entering an era of "killer microseconds" in data center applications [9]. Killer microseconds refer to μs-scale "holes" in CPU schedules caused by stalls to access fast I/O devices or brief idle times between requests in high throughput microservices. Whereas modern computing platforms can efficiently hide ns-scale and ms-scale stalls through micro-architectural techniques and OS context switching, they lack efficient support to hide the latency of μs-scale stalls. Simultaneous Multithreading (SMT) is an efficient way to improve core utilization and increase server performance density. Unfortunately, scaling SMT to provision enough threads to hide frequent μs-scale stalls is prohibitive and SMT co-location can often drastically increase the tail latency of cloud microservices.

In chapter II, I propose *Duplexity* [138], a heterogeneous server architecture that employs aggressive multithreading to hide the latency of killer microseconds, without sacrificing

the Quality-of-Service (QoS) of latency-sensitive microservices. Duplexity provisions *dyads* (pairs) of two kinds of cores: *master-cores*, which each primarily executes a single latency-critical *master-thread*, and *lender-cores*, which multiplex latency-insensitive throughput threads. When the master-thread stalls, the master-core borrows *filler-threads* from the lender-core, filling μs-scale utilization holes of the microservice. I propose critical mechanisms, including separate memory paths for the master-thread and filler-threads, to enable master-cores to borrow filler-threads while protecting master-threads' state from disruption. Duplexity facilitates fast master-thread restart when stalls resolve and minimizes the microservice's QoS violation. The evaluation results demonstrate that Duplexity is able to achieve 1.9$\times$ higher core utilization and 2.7$\times$ lower iso-throughput 99th-percentile tail latency over an SMT-based server design, on average.

1.2 μs-Scale Tail Latency

Managing tail latency is a primary challenge in designing distributed microservices. Queuing is a major contributor to end-to-end tail latency, wherein nominal tasks are enqueued behind rare, long ones, due to Head-of-Line (HoL) blocking. In chapter III, I investigate the tail latency problem in microservices and provide general guidelines for optimizing computer systems from a queueuing perspective to minimize tail latency [139]. In particular, I propose *Server Pooling* and *Common-Case Service Acceleration (CCSA)* as two key directions for minimizing the queuing delays and tail latency in microservices. Server Pooling is the basis of the framework introduced in the following chapter.

In chapter IV, I introduce *Q-Zilla* [141], a scheduling framework to tackle tail latency from a queuing perspective, and CoreZilla, a microarchitectural instantiation of the framework. On the algorithmic front, I first propose *Server-Queue Decoupled Size-Interval Task Assignment (SQD-SITA)*, an efficient scheduling algorithm to minimize tail latency for high-disparity service distributions. SQD-SITA is inspired by an earlier algorithm, SITA, which explicitly seeks to address HoL blocking by providing an "express-lane" for short tasks,

protecting them from queuing behind rare, long ones. But, SITA requires prior knowledge of task lengths to steer them into their corresponding lane, which is impractical. Furthermore, SITA may underperform an $M/G/k$ system when some lanes become underutilized. In contrast, SQD-SITA uses incremental preemption to avoid the need for a priori task-size information, and dynamically reallocates servers to lanes to increase server utilization with no performance penalty. Next, I introduce *Interruptible SQD-SITA*, which further improves tail latency at the cost of additional preemptions. Finally, I describe and evaluate *CoreZilla*, wherein a multi-threaded core efficiently implements ISQD-SITA in a software-transparent manner at minimal cost. CoreZilla is based on my earlier microarchitectural proposal, *Express-lane Simultaneous Multithreading (ESMT)* [140], which provides multiple physical execution lanes in an SMT core for different classes of task sizes, to run on virtual hardware contexts in an isolated manner, to prevent HoL blocking and minimize tail latency. By combining the ESMT microarchitecture and the ISQD-SITA scheduling algorithm, CoreZilla is able to improves tail latency over a conventional SMT core with 2, 4, and 8 contexts by 2.25×, 3.23×, and 4.38×, on average, respectively, and outperform a theoretical 32-core scale-up organization by 12%, on average, with 8 contexts.

1.3 Partial Service-Level Objectives

Service Level Objectives (SLOs) impose bounds on the average or tail of the end-to-end latency distribution in a cloud service, to ensure an acceptable level of service quality and user satisfaction. Auto-scaling frameworks, such as Google's Autopilot [171], continuously monitor the response time of the incoming requests to a service and upsize or downsize the service by increasing or decreasing the number of instances (VMs or containers) in the virtual cluster to meet the latency SLO at minimal cost [164]. However, with microservice-based applications, it is unclear which node in the microservice DAG needs to be scaled when end-to-end latency SLOs are violated or under-utilized.

In Chapter V, I propose Parslo—a Gradient Descent-based approach to allocate partial

SLOs to different nodes of a microservice DAG under an end-to-end latency SLO. Parslo isolates different microservice nodes within a DAG from one another and enables each microservice to be scaled independently through its own auto-scaling framework. At a high level, the Parslo algorithm breaks the end-to-end SLO budget into small "SLO units", and iteratively allocates one SLO unit to the best candidate microservice to achieve the highest total cost savings until the entire end-to-end SLO budget is exhausted. Parslo achieves the optimal solution, minimizing the total cost for the entire service deployment, and is applicable to general microservice DAGs, which may include microservice dependencies, branching path, as well as parallel indexing and sharding. My evaluation results demonstrate that Parslo reduces service deployment costs by more than $6\times$ in microservice-based applications, compared to a state-of-the-art SLO allocation scheme.

1.4 Mixed-Criticality Microservices

A key property of microservice-based architectures is that common microservices may be shared by multiple end-to-end cloud services. As an example, a speech-recognition microservice may serve as one of the first nodes in the microservice graphs of a variety of end-to-end services. However, given the dissimilarities in the orchestration and number of nodes in the microservice graphs across services, as well as varying end-to-end latency constraints, shared microservices may need to operate under differing latency constraints for each end-to-end service. As a result, in existing systems, most providers either deploy multiple instance pools for each latency constraint, or require all requests to needlessly meet the most stringent constraint.

In Chapter VI, I make a case that sharing microservice instances across multiple services can reduce significantly the number of instances, especially for deployments with highly asymmetric latency constraints. That said, I show that instance sharing is only beneficial if the arriving requests from each deployment class are scheduled intelligently across the execution resources within an instance, to meet their varying latency requirements. I propose

a request scheduling mechanism, called μ*Steal*, which leverages preemptive work and resource stealing to schedule the arriving requests to cores within an instance of a *mixed-criticality microservice*. μSteal provisions "core reservations" for each request class based on their latency requirements, but allows a class to steal cores from other classes if the cores would otherwise remain idle. Nonetheless, when a class requires its full reservation, μSteal preempts stolen cores, returning them to their reserved class. μSteal employs a runtime feedback controller augmented by a queuing theory-based analytical model to tune core reservations across classes, seeking to maximize the request throughput within each instance without violating any class's latency constraint. My evaluation results show that μSteal reduces the total number of instances required for several shared microservice deployments by 1.29$\times$ as compared to deploying multiple, segregated instance pools.

CHAPTER II

Duplexity: Enhancing Server Efficiency in the Face of Killer Microseconds

2.1 Introduction

We are entering the "killer microsecond" era in data center applications [9]. Due to advances in processor, memory, storage, and networking technologies, events that stall execution increasingly fall in a microsecond-scale latency range. Accesses to emerging storage-class memories [1, 33, 47, 136, 192, 157, 106, 62], rack-scale memory disaggregation [119, 46, 151, 65, 2], 100+ gigabit network communication [15], and accelerator/GPU micro-offloads [127, 22, 137] are example program activities that incur microsecond delays.

Lower latencies make it possible for data center architects to decompose monolithic applications into a collection of loosely-coupled microservices that interact over high-speed I/O to improve isolation, scalability, and maintainability [58]. Many cloud-based companies, including Amazon [188], Netflix [193], Gilt [216], LinkedIn [189], and SoundCloud [159] have adopted microservice architectures. Example microservices include content caching [56, 54], protocol routing [118, 150], key-value lookup [92, 142], query rewriting [8], or other steps performed across various application tiers [186]. Mid-tier microservices are particularly interesting objects of study since (1) they deal with both incoming and outgoing requests, (2) they must manage fan-out to leaf nodes and wait for

the responses, and (3) their computation typically takes only a few microseconds, which is often shorter than the delay waiting for leaves to respond [187]. However, as a consequence of shorter service times and higher throughputs, idle periods between requests also shrink to microsecond scales, even under moderate load.

Whereas contemporary computing systems are effectively equipped with mechanisms to hide nanosecond- and millisecond-scale stalls, they lack efficient support for microsecond-scale stalls [9]. Nanosecond-scale stalls are effectively hidden by microarchitectural mechanisms, such as Out-of-Order (OoO) execution and deep memory hierarchies, but these mechanisms are insufficient to hide microsecond-scale stalls. Conversely, operating systems use context switching to hide millisecond-scale latencies, such as when accessing disk. However, context switch overheads (5-20μs [114, 195]) are within the same latency orders as microsecond-scale stalls, so they are not a plausible latency-hiding technique for the microsecond regime.

Total cost of ownership (TCO)-conscious data center operators try to maximize performance per dollar by maximizing performance density and energy efficiency (throughput per unit area/power) [124, 107, 10]. Cycles wasted on microsecond-scale stalls or idle periods erode execution efficiency and increase TCO. User-facing workloads, such as web search, have strict latency objectives and time-varying load [10], thereby imposing the same characteristics to their underlying microservices. Nonetheless, data centers have myriad latency-insensitive scale-out applications (e.g., offline graph analytics) that can be flexibly scheduled to fill utilization holes during off-peak loads. Thus, a common way to improve server utilization is to co-locate latency-critical and batch workloads, allowing them to share resources [122, 39, 131, 213, 224].

Simultaneous multithreading (SMT) has been proposed to co-locate latency-critical and batch threads on the same core so that the batch threads fill the utilization holes caused by brief I/O stalls or inter-request idle periods [220, 215]. Already today, scale-out workloads deployed in data centers exhibit low CPU utilization due to lack of memory

level parallelism and front-end inefficiencies, calling for more SMT threads even in the absence of microsecond-scale stalls [55, 94]. As batch workloads also adopt mechanisms like storage-class memory or rack-scale disaggregation, these workloads, too, will incur such stalls. As a consequence, even more SMT threads must be added to ensure that, at any time, there are enough unstalled threads to fill a core's available execution bandwidth—the two threads offered by Intel's Hyper-Threading are not nearly enough.

Unfortunately, scaling SMT microarchitecture to support many more threads is prohibitive, due to high logic complexity, wire delay, limited register file (RF) capacity, and cache pressure/thrashing among threads. Moreover, as previous studies have shown [122, 215, 28], some SMT thread co-locations can have catastrophic impact on the tails of latency-critical threads, especially at high loads, due to contention for shared resources. To avoid compromising the tail latency of critical threads due to SMT interference, we instead design *Duplexity*, a server architecture that seeks directly to address the killer-microsecond challenge—to fill in the microsecond-scale "holes" in threads' execution schedules, which arise due to idleness and stalls, with useful execution, without impacting the tail latency of latency-critical threads.

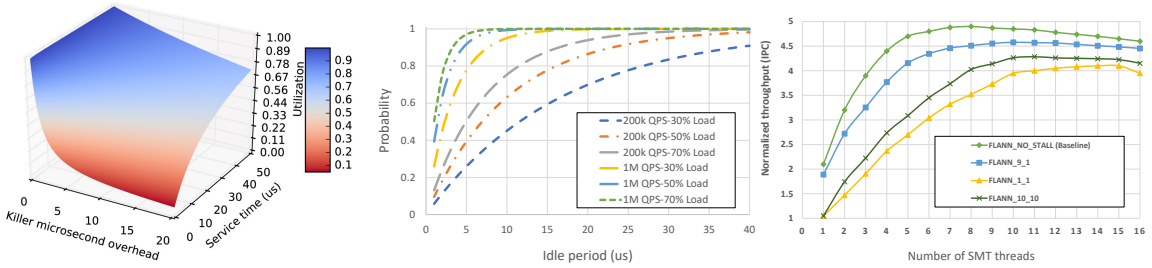
Duplexity is a heterogeneous server architecture that comprises two kinds of cores: *master-cores*—optimized for latency-sensitive microservices, and *lender-cores*—optimized for latency-insensitive throughput applications, which are arranged in pairs called *dyads*. Duplexity addresses microsecond-scale stalls by allowing master-cores to borrow threads from the lender-core in their dyad. Master-cores build on the concept of morphable cores [101] to switch between a single-threaded dynamically scheduled execution mode (when running the latency-critical *master-thread*) and a multi-threaded mode with in-order issue per thread (to fill in idle/stall periods with *filler-threads*). A key novel aspect of Duplexity is protection of the master-thread's micro-architectural state to maintain its QoS—filler-threads do not disrupt the caches, branch predictor, and other state held by the master-thread. When the master-thread becomes ready, Duplexity rapidly evicts filler-threads and grants the

master-thread exclusive use of the master-core.

Lender-cores employ a Hierarchical Simultaneous Multithreading (HSMT) architecture to maintain a backlog of *virtual contexts* that time-multiplex the lender-core's physical hardware contexts, and from which the master-core may borrow. We develop new mechanisms to support rapid transfer of virtual contexts into and out of the master-core. Overall, we seek to maximize performance density and energy efficiency (by increasing filler-thread throughput) while giving the master-thread nearly the performance it would enjoy running alone. Such a cooperative composition of cores yields Duplexity, a unique server architecture that is well-suited to the killer-microsecond era.

Our evaluation demonstrates that Duplexity can improve core utilization by 4.8× and 1.9×, and iso-throughput 99th-percentile tail latency by 1.8× and 2.7×, on average, over a baseline OoO and an SMT-based server architecture, respectively. Duplexity is the first server architecture that aims to improve server utilization in the presence of microsecond-scale stalls and idle periods, without sacrificing QoS and tail latency of microservices. In summary, we make the following contributions:

- We quantitatively explore the killer-microsecond challenge as it relates to the load of latency-sensitive microservices and show that microsecond-scale stalls arise due to both fast communication and brief idle periods.
- We show that conventional SMT is not a satisfactory solution as it may drastically harm tail latencies of microservices and cannot be scaled to hide microsecond-scale stalls.
- We propose Duplexity, a server architecture comprising highly multi-threaded and morphable cores that can borrow threads to recover cycles lost to microsecond-scale stalls and idle periods while providing isolation mechanisms to preserve QoS of latency-critical microservices.
- We compare Duplexity to other server designs. Existing alternatives either compro-



(a) (b) (c)

Figure 2.1: (a) Effect of μs-scale stalls on a closed-loop system, (b) Cumulative distribution of idle periods across various loads and service rates in an M/G/1 server, and (c) Throughput when varying the number of SMT threads for the FLANN workload on a 4-wide OoO core.

mise tail latency of microservices or fail to fully recover the cycles lost to microsecond-scale stalls.

2.2 Motivation and Background

We first motivate the problem space Duplexity seeks to address.

2.2.1 Killer Microseconds

With the advent of low-latency I/O in modern data centers, applications increasingly access data with single-digit microsecond latencies. For example, with state-of-the-art data center networking, a network round-trip at 40 Gbps can take only 2-4 μs [9]. At such latencies, RDMA-based disaggregated-memory systems [119, 46, 2, 151, 65] are expected to provide μs-scale remote memory accesses. Similarly, emerging memory technologies, such as 3D XPoint, have comparable access latencies [83]. Intel Optane SSD is an example low-latency storage device that builds upon 3D XPoint and enables 7-15 μs random block access [69]. At the higher-end of the microsecond spectrum, raw Flash can be accessed within tens of microseconds [191]. Fine-grain GPU/accelerator micro-offloads have similar latencies [127, 22, 137]. As a consequence, μs-scale stalls are quickly becoming a primary latency bottleneck, particularly as microservices replace monolithic data center applications.

Modern microarchitectures effectively hide ns-scale stalls caused by events like cache misses using instruction level parallelism and deep memory hierarchies. Modern operating

systems effectively hide ms-scale stalls caused by events like disk I/O accesses through context switches. However, existing mechanisms do not effectively hide μs-scale stalls or idle periods. Single-threaded deep speculation mechanisms like branch prediction and runahead execution [146] are not accurate enough to fill more than 10s of nanoseconds with future instructions from a stalled thread, and are inapplicable to fill idle periods. Similarly, prefetching techniques are at best able to hide the latency of cachable memory accesses (rather than general μs-scale I/O) and are not applicable to idle periods [30, 5, 4, 205, 181, 180]. Context switches themselves incur μs-scale overheads [114, 195], and are too expensive to amortize μs-scale stalls. Moreover, modern low-latency communication mechanisms rely on OS bypass interfaces (precisely to avoid latency of deep OS software stacks and their attendant caching inefficiencies) and hence are OS-transparent [90, 12, 151, 104, 162, 170, 103]. Alas, current warehouse-scale computers resort to spinning to maintain low latency despite μs-scale stalls, wasting these CPU cycles. It is for this reason that Google recently coined the term *killer microseconds* [9].

Killer microseconds due to stalls. We reason quantitatively about μs-scale stalls using a simple model. We consider a single-job closed-loop model representing a period of computation leading to a μs-scale stall event, such as a disaggregated memory access. The modeled system alternates between periods of computation and stalls. During stalls, CPU time is wasted, reducing utilization.

Figure 2.1(a) illustrates the utilization loss as we vary the length of stalls and the computation time between them. When stalls are short (left edge of front axis), utilization converges to 100%. So, for example, a DRAM-scale stall every few microseconds sacrifices an insignificant fraction of utilization. Correspondingly, when the computation interval between stalls is large (far edge of right axis), stalls reduce utilization only gradually. However, when stalls and computation periods are of a similar order, utilization drops precipitously, rapidly dropping towards 0% if stalls exceed the average computation interval (near corner). This model implies that, as the distance between μs-scale stalls shrinks, a

worrying fraction of CPU time may go to waste. Such compute-to-communication ratios are already being seen in mid-tier microservices that accept service-specific queries, fan them out to leaf microservers that perform relevant computations on their respective data shards, and then return the aggregated results [187].

Killer microseconds due to idleness. Utilization losses further mount due to idleness for microservices operating in the microsecond regime, even at moderate offered loads. To avoid long queuing delays, interactive services typically operate at 30-70% capacity [10]. Hence, idle periods are inherent. In ms-scale requests of monolithic services, idle periods correspondingly occur at ms-scale (e.g., around 1ms for a web search leaf service with a median service time around 4ms [133]). However, as faster I/O enables faster microservices, idle period time scales also shift.

Most cloud applications exhibit high service time variability and heavy-tailed service distributions [134, 71]. However, due to the memory-less property of Poisson request arrivals, idle periods of all M/G/1 queuing systems follow an exponential distribution, independent of the service distribution [72]; idle period duration is only a function of service rate and load. Figure 2.1(b) depicts the cumulative distribution of idle-period durations for M/G/1 microservices serving 200K and 1M Queries-per-Second (QPS) at offered loads of 30%, 50%, and 70% of capacity. As can be seen in the Figure, individual idle periods last only a few microseconds. For example, 200K and 1M QPS services at 50% load average idle periods of only 10μs and 2μs, respectively, despite being idle half the time. Existing mechanisms cannot exploit such short idle periods (indeed, they are too short even for hardware power management [132, 133, 121]).

2.2.2 Simultaneous Multithreading

Software/OS-based multi-threading is the typical approach for hiding millisecond-scale I/O stalls and improving resource utilization when a thread is blocked or idle for lack of work. However, software multi-threading is too coarse-grained to react at the microsec-

ond timescales of microservices. Simultaneous Multithreading (SMT), wherein each core multiplexes instructions from multiple hardware contexts, can increase instruction throughput and improve resource utilization. Several prior works use SMT to improve server utilization [220, 215] and SMT is widely reported to be enabled in modern data centers [94].

Not enough SMT threads. Unfortunately, the number of SMT threads supported by contemporary processors (typically, 2-4), is far too few to effectively hide frequent μs-scale stalls. To demonstrate that such stalls call for far more SMT threads, we consider a microservice benchmark based on FLANN [143], an open-source library for performing fast approximate nearest neighbor searches in high-dimensional spaces. FLANN uses Locality Sensitive Hashing (LSH) to perform k-nearest neighbor identification—a critical microservice employed in content-based similarity search. After an LSH lookup, the benchmark issues accesses to remote memory to retrieve remote objects indicated by the lookup. We modify the gem5 [14] simulator to intercept the remote accesses issued by the FLANN microservice and stall execution. The modified simulator draws stall durations from an exponential distribution; we vary the mean stall duration as a parameter in our experiments.

The computation FLANN performs between remote accesses varies with the number of LSH tables, buckets, and probes used in FLANN's lookup operation. We use these tuning knobs to adjust the interval between remote accesses. By adjusting the mean of the stall duration distribution and the interval between stalls, we model various killer microsecond scenarios. We consider four compute-to-stall ratios (9:1, 10:10, and 1:1; all in microseconds; denoted as FLANN-X-Y) and a configuration that does not stall (baseline). Note that while single–cache-line (64B) RDMA accesses take roughly 1μs [15], since we investigate the impact of stall durations on performance, we assume stalls to take 10μs and zero-latency in two of our workloads. Since we seek to analyze throughput (rather than latency, QoS, or queuing delays), in this experiment, we model a saturated queue of requests (i.e., 100% load; no idle period between requests) to only stall for remote accesses.

We measure normalized throughput as a function of the number of SMT threads, from one to 16. Figure 2.1(c) illustrates the resulting throughput on a 4-wide OoO superscalar core (we scale only the number of threads; we do not scale microarchitecture resources except provisioning additional architectural registers). The purpose of this experiment is to identify how many SMT threads are needed to saturate the 4-way OoO core. In the baseline with no stalls, 8 threads saturate the pipeline; more threads degrade performance due to interference. However, the workload variants with μs-scale stalls require more threads before performance gains level off. For example, the FLANN-9-1 workload (representing a 1 μs stall every 10 μs, for a 90% effective utilization) peaks at 11 threads, while the FLANN-1-1 (50% effective utilization) peaks at 15.

More threads increase interference and hurt QoS. Co-running latency sensitive threads with others can severely degrade tail latency and violate QoS requirements due to interference and cache pollution effects, even with only two SMT threads [28, 215, 122]. Nevertheless, we show that simply adding more threads is not a satisfying solution to fill μs-scale stalls, even if the only objective is to maximize instruction throughput. Note that in Figure 2.1(c) all three workloads with μs-scale stalls underperform the baseline. The frequent stalls in the these workloads result in more cache misses, as threads evict one another's data as their executions interleave. This effect is most apparent in the gap between the FLANN-10-10 and FLANN-1-1 workload: threads in both workloads are stalled 50% of the time, and yet the 10$\times$ more frequent stalls of FLANN-1-1 lead to much lower total throughput. Further note that the peak performance of FLANN-1-1 (at 15 threads) lags the peak performance of the baseline (achieved at 8 threads) by 16%; adding more threads cannot recover the throughput lost in FLANN-1-1's μs-scale stalls.

Unfortunately, there are daunting impediments to scaling SMT threads per core. First, more threads add L1 cache pressure. Cache capacity cannot increase without affecting cache hit time, which penalizes single-thread performance. Second, adding threads complicates fetch/dispatch/issue logic, prolonging its critical path and lowering clock frequency. Finally,

adding threads requires a larger register file to accommodate at least their architectural state. Again, scaling up this structure inevitably impacts wire delay and clock frequency—an effect we neglect in Figure 2.1(c) that would further exacerbate throughput loss. As a result, scaling SMT cores beyond 8 threads is ineffective for hiding μs-scale stalls, even when seeking only to maximize throughput.

2.3 Duplexity

We next present Duplexity—a server architecture that aims to fill in cycles lost to μs-scale stalls or idle periods while preserving tail latency and QoS. Duplexity comprises two kinds of cores: *master-cores*, optimized for latency-sensitive microservices and *lender-cores*, optimized for latency-insensitive scale-out (batch) applications. Duplexity addresses the killer microsecond challenge by borrowing "filler" threads from the lender-cores and executing them on the master-cores during the μs-scale "holes" arising from I/O stalls and idleness. To facilitate borrowing threads, master-cores and lender-cores are arranged in pairs, called 'dyads', with data paths that allow filler-threads running on the master-core to remotely access caches located at the lender-core. Master-cores build upon concepts from morphable cores [101], allowing them to morph between a single-threaded dynamically scheduled execution mode to execute their latency-sensitive *master-thread*, and a multi-threaded in-order execution mode to execute latency-insensitive filler-threads, borrowed from the lender-core. Lender-cores employ a Hierarchical Simultaneous Multithreading (HSMT) architecture, wherein they maintain a backlog of latency-insensitive threads that time-multiplex hardware contexts, from which the master-core may borrow. We integrate these concepts with efficient mechanisms to support rapid thread-context transfer into and out of the master-core and to protect the single latency-critical master-thread from interference by filler-threads. Our key objectives are (1) to fill in idle/stalled periods in the master-core with useful work from filler-threads, and (2) to minimize disruption, especially tail latency increases, of the master-thread.

There is a renewed interest in using simple, in-order cores for scale-out workloads [120, 86, 123]. However, simple cores incur a higher ratio of tail-to-average latency at large scales, and small configuration or parameter changes can result in large tail latency swings, making it difficult to achieve performance stability [28]. As such, latency-sensitive microservices with strict QoS targets are still typically run on OoO cores with advanced memory systems rather than a sea of scale-out-optimized simple cores [81, 40]. This dichotomy motivates the two operating modes of master-cores and our approach of coupling heterogeneous cores in dyads to facilitate thread borrowing.

When executing the master-thread, a master-core operates as an n-way OoO processor, with all execution resources dedicated to maximizing single-thread performance. However, whenever the master-thread becomes idle or incurs a μs-scale stall, the core's "morphing" feature is activated, which partitions the issue queue and register file and deactivates OoO issue logic to instead support InO issue of multiple filler-threads. The master-core then loads register state for these filler-threads from the lender-core's scheduling backlog and begins their execution. When the master-thread returns (stall resolves or new work arrives), it evicts the filler-threads, using hardware mechanisms that evacuate their register state as fast as possible. Minimizing performance disruption of the master-thread is challenging. In a key departure from prior work, we ensure that filler-threads cannot disrupt the cache state of the master-thread. We provision a path from the master-core's memory stage and front-end to the lender-core's caches; filler-threads access the memory hierarchy of the lender-core. Hence, when the master-thread returns, there is little evidence the filler-threads were ever there.

We first describe the microarchitecture of the lender-cores, as the master-core operates much like a lender-core when it operates in the multithreaded mode. We then explain our main contribution, the master-core, and how its microarchitecture enables adaptation between modes.

2.3.1 Lender-cores

The goal of lender-cores is two-fold: (1) support efficient multithreading for latency-insensitive scale-out workloads that nonetheless incur μs-scale stalls, and (2) lend threads to the master-core while it is stalling or idle. As we demonstrated in Section 2.2.2, the key requirement to hide μs-scale stalls is to provision more threads from which the lender-core can schedule. However, if too many threads are co-scheduled on the core, they will interfere with each other and may hurt performance. Hence, we suggest a Hierarchical Simultaneous Multithreading (HSMT) architecture with two levels of virtual/physical contexts, similar to Balanced Multithreading [197] and two-level warp scheduling in GPUs [148, 173]. Lender-core's datapath resembles an SMT core that supports as many threads as physical contexts, and has similar area costs and clock frequency, but, when a thread occupying a physical context faces a μs-scale stall, its architectural state is swapped with a ready virtual context to improve utilization and throughput.

By limiting the number of active threads, the lender-core prevents performance degradation or diminishing returns (as observed in Figure 2.1(c)) due to interference of many threads, yet it virtually enables sufficient threads to hide μs-scale stalls. We find 8 threads as the sweet spot for the number of physical contexts for three main reasons: First, as we showed in Section 2.2.2, the core's throughput saturates around 6-8 threads and may even drop beyond 8 threads in the absence of μs-scale stalls. Even with stalls, due to interference, the core is not able to match the throughput it achieves without stalls if many threads are co-scheduled. Second, as illustrated in Figure 2.2(a) and shown by prior work [101, 79, 178], the gap between OoO and InO issue vanishes at $\sim$8 threads. Hence, we can employ an InO datapath to reduce precise state and pipeline flush complexity and avoid area/energy costs of OoO structures, especially since the lender-core targets only latency-insensitive threads. Finally, while up to 8-thread SMT designs are commercially available [172], building a core with more than 8 physical SMT contexts may be impractical due to logic/wire complexity and register file constraints.

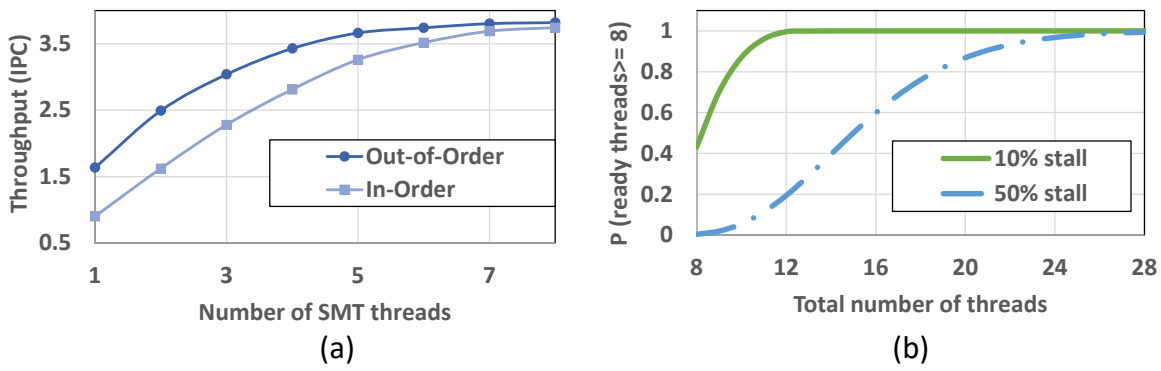


Figure 2.2: (a) Throughput of multi-threaded SPEC workload mixes for varying InO/OoO SMT threads on a 4-wide OoO core. (b) Probability of having at least 8 ready threads under varying thread counts and stall rates.

We develop a simple analytic model to determine how many virtual contexts are needed to fill eight physical contexts as a function of the fraction of the virtual thread stall time. The distribution of ready threads is then given by a Binomial $k \sim \text{Binomial}(n, 1-p)$, where k represents the number of ready threads, n the number of virtual contexts, and p the probability a thread is stalled. We plot $P(k \geq 8)$ as a function of n for two stall probabilities in Figure 2.2(b). When threads are stalled only 10% of the time, 11 virtual contexts are sufficient to keep the 8 physical contexts 90% utilized. However, when threads are 50% stalled, 21 virtual contexts are needed. As a result, the number of required virtual contexts may be different depending on the workload.

A lender-core's microarchitecture is shown in Figure 2.3. The datapath is identical to an 8-threaded InO SMT. We note that this core is quite simple and area-efficient, since it does not require any OoO execution logic. The lender-core's front-end maintains a pointer to a FIFO run queue in dedicated memory, which holds the state of all virtual contexts. When a physical context stalls, its context is dumped to the tail of the run queue. Then, another context's architectural state is loaded from the run queue. The length of the run queue is not limited by hardware, as the number of required virtual contexts may vary. OS/cluster-level scheduling frameworks must provision enough threads to each lender-core to ensure the core is fully utilized and threads do not starve.

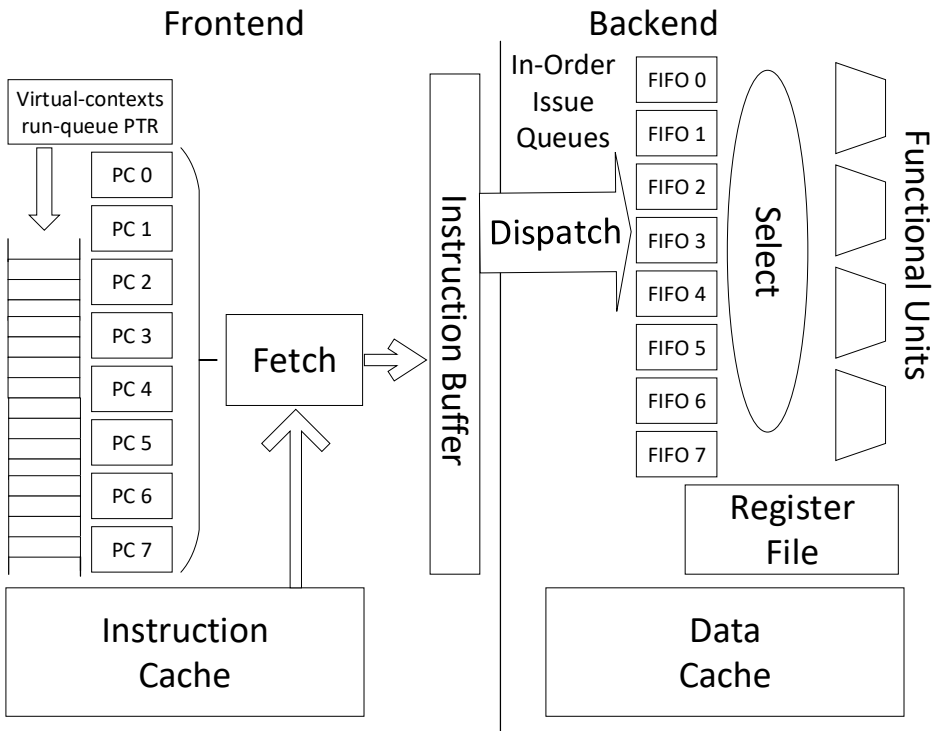


Figure 2.3: Lender-core: 8-way InO Hierarchical SMT (HSMT).

Master-cores borrow threads from a lender-core by stealing a virtual context from the head of its run queue. The master-core and lender-core in each dyad share the dedicated memory region where virtual contexts are stored. Additional challenges arise when filler-threads access memory; we defer discussion of these to Section 2.3.2.3.

2.3.2 Master-cores

As discussed in Section 2.2.2, co-running additional threads alongside a latency-sensitive thread can drastically harm tail latency [122, 215, 28]. As such, many prior works reject SMT for latency-critical applications (e.g., [122, 28]) and most server-optimized scale-out processors, such as Cavium ThunderX [23] and Qualcomm Centriq [165], do not employ multithreaded cores. However, in the microsecond regime, where threads frequently face long stalls, SMT provides the most promising approach to recover the lost cycles.

This dichotomy motivates our design for master-cores, where we execute the *master-thread* by itself, to preserve its tail latency, but multiplex several *filler-threads* during

master-thread stalls, to recover throughput. In master-thread mode, the master-core operates as a single-threaded 4-wide OoO superscalar core, optimizing for single-thread performance and minimal tail latency. In filler-thread mode, while the master-thread is stalled/idle, the master-core switches from its single-threaded OoO issue mechanism to the InO HSMT mechanism of a lender-core. It then multiplexes multiple filler-threads to use the available issue bandwidth. Together, these modes maximize performance density and energy efficiency by maximizing executed instructions.

2.3.2.1 From MorphCore to Master-core

Our master-core microarchitecture builds upon *MorphCore* [101]. MorphCore rests on two insights: (1) a 6-8 way in-order SMT core can achieve better total throughput than a single-threaded OoO core and (2) such an in-order core requires a subset of the hardware mechanisms already present in an OoO core. MorphCore reuses most hardware structures (instruction buffer, ALUs, RFs, load/store unit, etc.) in both execution modes. In multi-threaded mode, it partitions instruction buffers, reservation station (into multiple in-order issue queues), and the reorder buffer among threads, which all share functional unit pipelines. It repurposes the core's physical register file as architectural registers for each thread. Finally, it disables register renaming, dynamic scheduling, and the load queue to save energy. On a mode switch, MorphCore swaps the extra threads' architectural registers from a dedicated memory region using microcode.

We describe our master-core design by starting with MorphCore as an initial strawman and successively addressing challenges that arise in the killer microseconds context. First, we replace the conventional in-order SMT operating mode of MorphCore with the HSMT architecture of the lender-cores, described in Section 2.3.1. Thus, when running filler-threads, the master-core will have sufficient available virtual contexts to hide killer microsecond stalls.

Second, we alter several aspects of how MorphCore transitions modes. The master-core

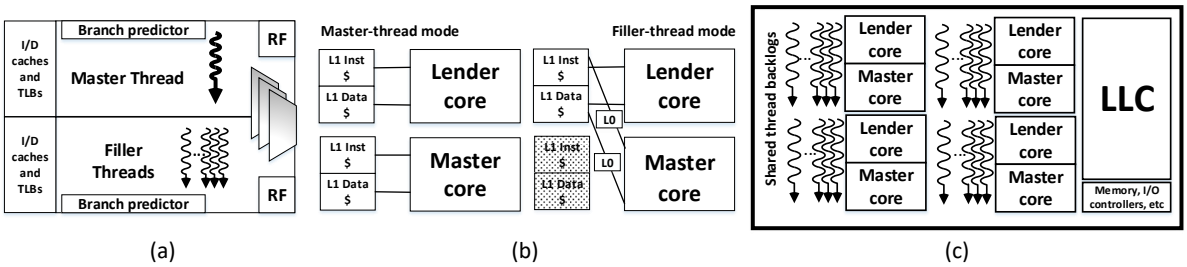


Figure 2.4: (a) A naive master-core design where stateful micro-architectural components are replicated across modes, (b) A Duplexity dyad composed of a master-core and a lender-core, and (c) Layout of a Duplexity server processor chip.

triggers a transition whenever the master-thread becomes idle or incurs a μs-scale stall. We drain instructions elder than the stalling instruction and flush younger instructions. In contrast to MorphCore, a master-core does not evict the architectural register state of the master-thread; it retains its registers to facilitate fast restart when the stall resolves, after which all in-flight instructions from filler-threads are immediately squashed.

There are two challenges with this strawman master-core design, if used alone. First, filler-threads thrash the cache, TLB, and branch predictor state of the master-thread. When the master-thread resumes, it will incur many cache misses, which may adversely affect its tail latency. Second, filler-threads have no guarantee when they will be scheduled on the master-core; they are only scheduled when the master stalls, and hence they may starve. We next solve these problems.

2.3.2.2 Segregating State

We must ensure that filler-threads do not thrash the master-thread's state. The naive approach is to replicate all stateful micro-architectural structures (register files, caches, branch predictor, TLBs, etc.), segregating the filler-threads' from the master-thread's state. We compare against this alternative, shown in Figure 2.4(a), in our evaluation. The problem with replicating all structures is that caches and register file are large and power-hungry. In particular, depending on microarchitecture, register files usually consume 5%-20% and L1 caches consume 10%-40% of a core's area [117]. So, this approach undermines Duplexity's performance density and energy efficiency objectives.

Instead, Duplexity replicates only the area-inexpensive structures. We provision a full-size TLB and reduced-size branch predictor for exclusive use by filler-threads. For the register file, we provision empty physical registers to store the architectural state of filler-threads, using the renaming logic to track the assignment of logical filler-thread registers to physical registers. Once its in-flight instructions are squashed or drained, the master-thread occupies only enough physical registers to maintain its architectural state.

To avoid replicating caches, we introduce the concept of dyads, which we discuss next.

2.3.2.3 Master-Lender Dyads

Instead of replicating caches, we pair a master-core with a lender-core to form a *dyad*. When a master-core morphs into filler-thread mode, the filler-threads remotely access the L1 instruction and data caches of the lender-core. The dyad provides data paths from the master-core's fetch and memory units to the lender-core's caches, as shown in Figure 2.4(b). This approach has two benefits: (1) it protects the master-thread's state, and (2) it allows filler-threads to hit on their own cache state as they migrate between the cores. However, this approach also entails two challenges: (1) The L1 access latency of filler-threads on the master-core is $\sim$3 cycles higher than local cache access in either core. (2) The capacity pressure and bandwidth requirements on the lender-core's caches increase, since both cores may access them.

We address these challenges by provisioning a small 2KB L0 I-cache and a 4KB L0 write-through D-cache in the master-core for accesses to the lender-core's L1 caches. Although these L0 caches have low hit rates, they act as effective bandwidth filters and service many sequential accesses, especially for instructions. Whereas capacity pressure on the lender-core's L1 cache is high, HSMT is inherently latency-tolerant; our evaluation demonstrates a net throughput win. The lender-core L1 D-cache maintains inclusion with L0 D-cache and forwards invalidations to maintain coherence.

2.3.2.4 Fast Filler-thread Eviction

A key Duplexity objective is to ensure fast master-thread resumption when it becomes ready. We use several approaches to accelerate restart.

First, we reuse the L0 data cache to accelerate spilling filler-thread architectural state. The L0 cache is write-through, hence, its contents can be discarded or overwritten at any time. When the master-thread becomes ready, all pending filler-thread instructions are immediately flushed. Then, all physical register file read ports are used to read filler-thread architectural state and write it to the L0 data cache. With 8 read ports and an L0 write bandwidth of one cache-line per cycle, it takes less than 50 cycles to spill the filler-threads. We assume each thread requires 16 64-bit GP integer registers and 16 128-bit XMM floating-point/SIMD registers, per the x86-64 ISA. The master-core's physical register files include 144 registers—sufficient for architectural registers of 9 threads (the master and 8 filler-threads). The 4KB L0 capacity is sufficient to absorb the spill of all filler-thread registers.

During the spill, the master-core can begin dispatching master-thread instructions but instructions do not issue until read ports become available. As the master-thread's cache state is intact, fetches are likely to hit. Furthermore, the master-thread's architectural state is already present in the physical register file, as we do not evict it. Filler-thread register state is drained from the L0 to the dedicated backing store in memory in the background. In short, master-thread resumption incurs roughly a 50-cycle delay.

2.3.3 Summary

Figure 2.4(c) depicts the final Duplexity design, comprising several dyads each with a master- and a lender-core that share virtual contexts. The lender-core uses HSMT with 8 physical contexts sharing an 8-way InO datapath. HSMT enables the lender-core to hide μs-scale stalls in its latency-insensitive virtual context pool. The master-core can fill the master-thread's μs-scale holes with filler-threads borrowed from the lender-core by morphing into an InO HSMT architecture, while still protecting the master-thread from tail

latency disruption. Sharing virtual contexts across the dyad prevents contexts from starving.

2.4 Discussion

Scheduling. Duplexity affects several aspects of how the OS must manage SMT threads. The OS must schedule latency-critical threads on master-cores and provision the virtual contexts for each dyad. Since the number of virtual contexts is variable, and should be tuned based on the frequency and duration of stalls, a dyad appears to software as if it supports a variable number of hardware threads. The scheduling of virtual contexts on the physical contexts of master- and lender-cores is transparent to software. Conceptually, the master-core is exposed to software as a single-threaded core, while virtual contexts belonging to a dyad belong to the lender-core. Existing CPU hot-plug mechanisms [147] may be applicable to vary the number of virtual contexts at runtime. Alternatively, OS designs like Barrelfish [218], which separates core, thread, and OS abstractions, might be adopted.

The OS must select how many virtual (filler) contexts to activate in a dyad. One option is to simply over-provision, but this may lead to long scheduling delays for ready virtual contexts. Alternatively, a data-center-scale scheduling layer might optimize thread assignments via data-center-wide optimization [39, 38]. We find empirically that 32 virtual contexts per dyad are sufficient to hide stalls in our most pessimistic scenarios, wherein both the latency sensitive and batch threads incur frequent stalls (1 μs stall per μs of compute). If batch threads do not incur μs-scale stalls, 16 batch threads are sufficient; eight each to fill contexts on the lender and master-cores. If only batch threads incur μs-scale stalls (and thus never run on the master-core), 21 threads are sufficient to occupy the lender-core (see Figure 2.2(b)).

We use a simple round-robin scheduling policy for virtual contexts, which is easy to implement in hardware and provides some fairness/starvation avoidance. Virtual contexts are scheduled on a physical context for a 100 μs quantum to prevent starvation. Because this

quantum is far lower than the OS scheduling quantum, the two scheduling mechanisms do not interfere—from the OS perspective, all virtual contexts are active, much like hardware threads in an SMT system. Unused virtual contexts are parked via HLT, much like unused hyperthreads. Note that the two-level scheduling applies only to latency-insensitive batch threads.

Throughput threads. Duplexity's approach increases the number of active threads per chip. The need for more threads to maintain utilization is an inherent consequence of more frequent and longer stalls and is a key aspect of scale-out server architectures. For multi-programmed workloads, a possible consequence is an increase in server memory capacity requirements. Duplexity's improved latency tolerance dovetails with emerging memory technologies like 3D XPoint [83, 1] which trade improved capacity for longer access latency. For many classes of scale-out batch workloads (e.g., graph analytics [128], task-parallel applications [17, 168], Spark [217], and Hadoop [206]), it is often possible to partition data shards or tasks among threads at finer granularity to exploit more parallelism within the same memory footprint and provide flexibility in the number of threads. Moreover, individual tasks are often latency insensitive, making them well-suited to Duplexity. These workloads typically benefit substantially from hardware multithreading as they can overlap multiple remote accesses and provide (remote) memory-level parallelism (MLP) through thread-level parallelism (similar to the execution model of GPUs). In the absence of sufficient hardware threads, such distributed big-data algorithms must rely on complex asynchronous programming models and continuation/call-back mechanisms to provide MLP [187].

Duplexity protects the master-thread from interference by filler-threads. Nevertheless, batch/filler-threads may interfere with one another. Existing work on intelligent co-location may be applicable to mitigate such interference [213, 39, 224].

Demarcating stalls. A second aspect of Duplexity is that we assume that hardware can recognize the start and end of μs-scale stalls. For example, remote disaggregated memory accesses can be recognized from their memory translations or use queue pair-based memory

models [151] that bypass the kernel, as in other forms of polling-based high-performance I/O protocols that are transparent to the OS [90, 162, 12, 104, 170, 103]. Stalls end when remote loads return. Alternatively, special monitoring instructions (e.g., `mwait`, variants of `hlt` [66]), can wake upon cache coherence activity or data/work arrival [126].

Alternative approaches. GPUs and user-level multithreading present two alternative strategies to hide μs-scale stalls by multiplexing many threads. GPUs employ large register files to accommodate all active threads and accelerate context switching [87, 100, 50]. However, GPUs are applicable only to workloads amenable to their distinct programming model (CUDA/OpenCL), which is typically ill-suited for most software frameworks that run in the cloud—especially I/O intensive workloads or those where concurrency arises from request rather than data parallelism. User-level multithreading (e.g., [30, 44, 18]) enables fast context-switching through cooperative threading. This approach also entail substantial software re-engineering and does not apply to existing binaries. Both of these approaches are better suited for throughput rather than latency-sensitive applications. Moreover, neither approach protects a latency-critical thread from throughput-thread interference.

2.5 Evaluation Methodology

We use gem5 x86-64 [14] to evaluate Duplexity. We extend gem5 to model the master-core (and our other OoO baselines) and evaluate its performance in detailed simulation. We model a single dyad. For the scale-out workloads running on filler-threads, we determine the throughput of multi-threaded workloads on the in-order master-/lender-cores through trace-based simulation. We analyze energy and area with McPAT [117], and apply the changes described in [210] to more accurately model OoO cores. We estimate tail latencies using the BigHouse [134] methodology. We simulate the queuing system until we achieve 95% confidence intervals of 5% error in reported results. We measure IPC in gem5 and use it to determine the service rate of an FCFS M/G/1 queuing system. We then simulate the high-level behavior of the queue at request (rather than instruction) granularity. The

M/G/1 assumption is in line with prior studies [220, 97, 82]. We generate service times in BigHouse by measuring their distribution on real hardware, and scaling them using IPC slowdowns measured in gem5.

Overheads. The master-core builds upon a 4-wide OoO microarchitecture. We add the ability to transition to filler-thread mode, much like MorphCore. As such, the master-core entails all the hardware overheads of MorphCore (extra muxing paths in the front-end, select, and wakeup logic, and additional bypass paths in the back-end). Khubaib reports an area overhead of $\sim$2% for these structures [101]. In addition, the master-core provisions a TLB, reduced-size branch predictor, L0 I/D caches for use by filler-threads, and fetch/memory-access data-paths to the lender-core's caches. We model these additional structures with McPAT and find that the additional TLBs, branch predictors, and L0 caches impose area overheads of 0.7%, 1.2%, and 1%, respectively. The total area overhead of the master-core is approximately 5% compared to a baseline 4-wide OoO core. The static power overhead is within 5% of the baseline. In contrast, a master-core variant that replicates all stateful structures, including L1 caches, incurs a 38% area overhead. Our master-core requires additional multiplexers at various pipeline stages to mux between InO/OoO data paths used in different modes. Assuming 20 gates per pipeline stage [29], we estimate a cycle time penalty of 4% for these muxes. We include area, frequency, and power overheads in our results.

Workloads. We consider the following microservices, two of which are simplified/simulator-friendly versions of microservices from [186]; the other two are constructed using the same framework.

- **FLANN:** We evaluate two configurations of the FLANN [143] microservice introduced in Section 2.2.2; *FLANN-HA* (High-Accuracy) has an LSH lookup latency of 10μs and identifies a large number of nearest-neighbor candidates. *FLANN-LL* (Low-Latency) reduces lookup latency to only 1μs by using longer hash keys. Both of these configurations issue a one-sided single–cache-line remote access to retrieve

one of the identified nearest neighbors. We assume single–cache-line RDMA read latency to be exponentially distributed with a 1μs average [15].

- **Remote Storage Caching (RSC):** We implement a remote storage caching microservice; a simplified variant of Flash caches [21, 105, 3, 80]. Our RSC microservice maps linear block addresses of a remote storage system to a local low-latency SSD using Cuckoo hashing [155]. We only consider read transactions; allocation and coherence mechanisms fall outside the scope of our experiments. Look-up latency is 3μs, which, upon a hit, is followed by 8μs average access latency to Intel's Optane SSD [69] through user-level polling [191] and 4μs average latency for a 4KB *memcpy*. Though optimistic for current-generation Optane, we believe these characteristics are representative of future devices.
- **McRouter:** We employ a consistent hashing microservice based on Facebook's McRouter [118, 150]. This microservice routes Key-Value (KV) operations to 100 leaf servers via a consistent hash function and synchronously waits for leaf responses. We consider a state-of-the-art RDMA-based low-latency KV store that uses single-sided operations to minimize communication latency [92, 93]. The root microservice requires 3μs to route each request and the leaf KV store requires 3-5μs depending on the KV operation [92].
- **Word Stemming:** Stemming is a normalization process used to reduce words to their root and is a core query rewriting microservice employed in various cloud applications, such as web search. We develop a word stemming microservice based on Oleander's implementation of the Porter stemming algorithm [160, 161]. This microservice incurs no μs-scale stalls, since it is a leaf service. Hence, core under-utilization arises only due to the idle time between requests. Furthermore, it is state-less; it hard-codes all stemming paths (prefixes, suffixes, etc.) into the program control-flow. It requires an average processing time of 4μs.

Filler-threads execute distributed PageRank and Single-Source Shortest Path algorithms based on bulk synchronous processing [199] and synchronous queue pair-based disaggregated memory model [151] on a single dataset representing a subset of the Twitter graph [112]. Reading a remote vertex requires a single–cache-line RDMA read that takes 1μs [15]. Since almost half of vertices are accessed remotely through RDMA, our filler-threads also require 1μs stall time per each 1-2μs of compute. We execute 32 filler-threads per dyad.

Design Configurations. We compare a Duplexity dyad to a variety of alternative core microarchitectures. Our performance density and energy efficiency studies pair each core alternative with a throughput-oriented HSMT core (configured to match Duplexity's lender-core) for a fair throughput comparison. Our main objective is to contrast the impact of these architectures on the microservice's tail latency/QoS.

We consider the following alternatives:

(1) Baseline: A 4-wide OoO core that only executes the latency-sensitive microservice.

(2) SMT: Baseline augmented with a second SMT batch thread, using ICOUNT [196]. The core does not prioritize the latency-critical thread.

(3) SMT+: Similar to SMT but prioritizes the latency-sensitive microservice over its co-runner unless the microservice thread is stalled. For bandwidth (per-cycle) resources (Fetch, Issue, Commit), SMT+ always prioritizes the latency-sensitive thread and only allocates slots to the co-runner if the microservice thread does not need them [45]. For storage resources (IQ, ROB, LSQ), SMT+ limits the co-runner to occupy at most 30% of the slots [167].

(4) MorphCore: MorphCore as proposed in [101], running 8 filler-threads when it morphs.

(5) MorphCore+: MorphCore extended with our HSMT mechanism and paired with a lender-core (i.e., it borrows threads from a shared virtual context pool, like a master-core).

(6) Duplexity + replication: A Duplexity (master-core+lender-core) variant wherein all master-core's stateful structures, including caches, are replicated.

(7) Duplexity: Our final Duplexity (master-core+lender-core) design. (Master-core shares

Table 2.1: Microarchitecture details of Duplexity

Baseline/SMT	4-wide OoO, 144-entry ROB/PRF, 48-entry LQ, 32-entry SQ, ICOUNT fetch for SMT Tournament predictor: bimodal (16K), gshare (16K) and selector (16K); 32-entry RAS; 2K-entry BTB, 64-entry I/D TLBs
Lender-core	8-way InO HSMT, 32 virtual contexts, 4-wide issue, 128-entry ARF, Round-Robin fetch, gshare (8K) predictor, 2K-entry BTB, 64-entry I/D TLBs
Master-core	Transitions between singe-threaded OoO and InO HSMT, uarch same as baseline; tournament(16k)/gshare(8k), separate TLBs for the two modes, 2KB/4KB I/D write-through L0 caches
L1 caches	Private 64KB I/D, 64B lines, 2-way SA
LLC	1 MB per core, 64B lines, 8-way SA
Memory	50 ns access latency
NIC	FDR 4x Infiniband (56Gbit/s, 90M ops/s)

Table 2.2: Area and clock frequencies of different design configurations

Component	Area	Frequency
Baseline OoO	12.1 mm^2	3.4 GHz
SMT	12.2 mm^2	3.35 GHz
MorphCore	12.4 mm^2	3.3 GHz
Master-core	12.7 mm^2	3.25 GHz
Master-core + replication	16.7 mm^2	3.25 GHz
Lender-core	5.5 mm^2	3.4 GHz
LLC	3.9 mm^2/MB	N/A

L1 I/D caches with its neighbor lender-core when running filler-threads; L0 caches as bandwidth filters/register-buffers).

We report microarchiture configuration details in Table 4.1 and area/frequency results, obtained using McPAT [117] and CACTI [145] for 32nm technology, in Table 2.2.

2.6 Efficiency Results

2.6.1 Core Utilization

Figure 2.5(a) reports average core utilization. We calculate core utilization by dividing the number of retired instructions per cycle by the core's peak retire bandwidth (i.e, 4).

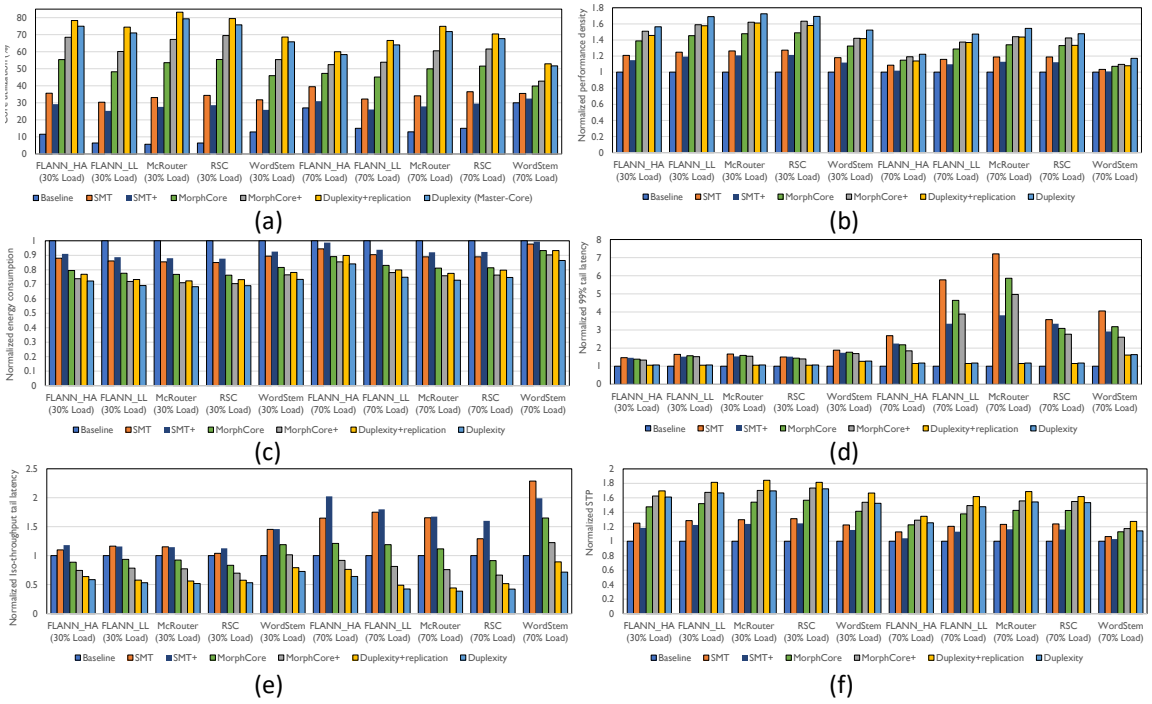


Figure 2.5: (a) Core utilization, (b) Normalized performance density, (c) Normalized energy consumption, (d) Normalized 99% tail latency, (e) Normalized iso-throughput 99% tail latency (f) Normalized system throughput (STP) for batch threads.

These results include only the utilization of the master-core or its alternatives. Whereas instructions executed from borrowed threads are included, instructions executed on the lender-core are not. Baseline OoO core utilization is at most 29% and drops to 5.7% when load is low (30%) and the stall ratio is high (e.g., ∼60% in McRouter). The baseline OoO scheme is single-threaded and has no mechanism to mitigate μs-scale stalls, so this result is not surprising. All other architectures improve utilization by executing instructions from filler-threads. SMT and MorphCore yield considerably lower utilization than HSMT-based designs (MorphCore+ and Duplexity variants) as 8 filler-threads are insufficient to hide stalls.

The Duplexity variants achieve the highest utilization. However, as load or work/stall ratio increases, utilization falls since filler-threads execute instructions only when the master-thread is idle/stalled. When active, the master-thread runs by itself and utilization depends

only on the master-thread's IPC; prior work [55, 94] reports that scale-out applications exhibit low ILP/MLP and fail to fully utilize a core. Conversely, at low load or work/stall ratio (e.g., McRouter and RSC at 30% load), Duplexity fills wasted cycles with useful work from filler-threads and increases issue bandwidth utilization to 79%. Finally, whereas Duplexity improves average utilization by 4.8× and 1.9× over the baseline and SMT, respectively, it always achieves lower utilization (3.6%, on average) than Duplexity + replication. Replication reduces lender-core cache pressure when the master-core runs filler-threads. Nevertheless, replicating caches is area-inefficient and incurs a drastic performance density penalty (see Table 2.2).

The low utilization of MorphCore+ compared to Duplexity arises because of (1) cold misses when the latency-critical thread resumes execution (due to cache pollution by filler-threads) and (2) mode switching latency. SMT+ achieves the lowest utilization (except for the baseline OoO) as it limits the co-runner thread to use only 30% of hardware resources to minimize master-thread interference; it achieves 2.4× lower utilization, on average, compared to Duplexity. The WordStem microservice provides the least opportunity for utilization improvement as it does not incur μs-scale stalls; opportunity arises only during idle periods. However, even under 70% load of WordStem, Duplexity improves issue bandwidth utilization by 69% and 41% compared to baseline and SMT, respectively, because the IPC of 8 co-running filler-threads is substantially higher than OoO and SMT.

2.6.2 Performance Density & Energy Efficiency

Performance density (Figure 2.5(b)) and energy efficiency (Figure 2.5(c)) are widely used in the literature to normalize performance over cost and estimate TCO [124, 222]. In these results, we pair alternative core variants with a throughput-oriented InO HSMT core (configured to match Duplexity's lender-core) to compare throughput fairly.

Performance density—instructions retired per unit-time per unit-area—enables comparison of area efficiency, which is critical when comparing TCO across heterogeneous

designs [124]. Figure 2.5(b) reports normalized performance density across design points. Duplexity achieves the highest result: 49% and 28%, on average (up to 72% and 37%), higher than baseline and SMT, respectively. These results generally track utilization results (Figure 2.5(a)) with two exceptions: First, the gaps between designs are smaller, since we normalize by the area of an entire chip, including the shared LLC. The throughput core (i.e., lender-core) and LLC area mask differences in core efficiency. Second, while Duplexity + replication yields the highest core utilization, its performance density is, on average, 9.2% (up to 13.4%) lower than Duplexity due to the large area overhead of replication.

Although it achieves slightly higher utilization, Duplexity + replication remains an undesirable design point. When the master-core does not borrow threads, all designs except Duplexity + replication achieve roughly the same performance density as the baseline. However, Duplexity + replication loses $\sim$17% density relative to the baseline—due to its considerably higher area—which translates to higher TCO. The replication cost is even higher if we apply Duplexity to scale-out processors [124] (e.g., Cavium ThunderX [23]), which trade SRAM for cores and share a modestly sized LLC (e.g., 8-16MB) across many cores (e.g., 32-64 cores).

To measure energy, we divide the power consumption of each design by the average number of instructions retired each cycle. Figure 2.5(c) reports normalized energy results, which largely mirror the trend of performance density. Duplexity nearly always consumes the least energy, as Duplexity is able to retire the highest number of instructions per cycle among all designs except Duplexity + replication, which falls short on energy-efficiency because it replicates power-hungry structures. In particular, Duplexity is able to reduce energy consumption by 34% and 21%, on average, compared to baseline and SMT architectures, respectively.

Duplexity replicates some units (e.g., TLBs, predictors), which are not used when the master-thread executes. At first blush, this may appear antithetical to our goal of maximizing utilization. We decide whether to replicate or borrow a structure based on area and power—

we only replicate inexpensive structures, for which replication provides a performance density and energy efficiency win. The metrics measured in Figures 2.5(b) and 2.5(c) capture this trade-off; if replication is area- or power-inefficient, these metrics worsen. In other words, these metrics represent the overall system utilization normalized against the area/power cost of all units.

2.7 Performance & QoS Results

We report QoS via aggregate throughput for batch threads and tail-latency for the latency-critical microservices.

We determine Duplexity's impact on tail latency as described in Section 2.5. Even small service time increases are amplified in the tail by queuing effects, especially at high loads [99]. Figure 2.5(d) reports the normalized 99th percentile tail latency under various load levels. Whereas SMT, Morphcore, and Morphcore+ increase tail latency by up to 7.2×, 5.8×, and 4.9×, respectively, Duplexity only increases tail latency by 19%, while recovering 4.8× higher core instruction throughput.

We make two further observations from Figure 2.5(d): First, SMT+ usually achieves considerably (up to 89%) lower tail latency than SMT as it minimizes the master-core interference via prioritization and partitioning. In fact, when the mode switch frequency is high and stalls are short, SMT+ achieves lower tail latency than MorphCore/MorphCore+, as it is less disruptive to master-thread issue bandwidth. However, its tail latencies are still higher than the baseline (up to 3.8×), due to interference on caches and core resources. Second, while WordStem does not incur μs-scale stalls and does not maintain any state across requests, it nevertheless is sensitive to instruction cache interference and suffers high tail latencies under SMT and MorphCore variants.

Figure 2.5(e) reports normalized iso-throughput 99% tail latency of the master-thread across workloads and load levels, to make a system-level assessment. These results normalize across designs such that they achieve the same cost by varying input load in proportion to

the performance density reported in Figure 2.5(b). The intent of this metric is to compare the impact of two microarchitectures on tail latency at a particular throughput while accounting for the fact that the designs differ in area, and therefore cost. Improving this metric implies a more tail-tolerant microarchitecture at a given cost. Duplexity achieves the lowest iso-throughput tail-latency, which is up to 2.6× and 4.3× (1.8× and 2.7×, on average) lower than iso-throughput tail latencies achieved by baseline and SMT, respectively. Whereas MorphCore variants achieve lower iso-throughput tail latencies over the baseline due to their high efficiencies, SMT variants lengthen iso-throughput tail latency compared to the baseline, as they do not sufficiently improve utilization. Perhaps surprisingly, while SMT+ provides isolation and prioritization mechanisms to protect the master-thread's performance, in some cases it yields worse iso-throughput tail latencies than SMT (up to 23%), due to lower utilization.

For the batch threads, we report system throughput (STP) [51], a metric that considers both performance and fairness for multi-threaded workloads. Figure 2.5(f) reports normalized batch-thread STP. Again, we pair all cores with an HSMT core for a fair comparison. MorphCore+ and Duplexity + replication yield better STP than Duplexity, because Duplexity shares the lender-core's caches between the master-core and the lender-core when the master-thread is idle/stalled, degrading performance. Nevertheless, Duplexity still improves batch STP over the baseline and SMT by an average of 52% and 24%, respectively, within 8% of the best STP achieved by Duplexity + replication.

2.8 Case Study: Interconnect Utilization Analysis

Duplexity improves CPU utilization via thread-level parallelism; thus, it requires enough busy threads to be effective. As such, it is critical that bottleneck resources provide sufficient bandwidth to support all threads, otherwise, we simply shift from one bottleneck to another. For disaggregated memory, interconnect bandwidth is a key resource. So, we confirm that Duplexity's interconnect requirements are feasible.

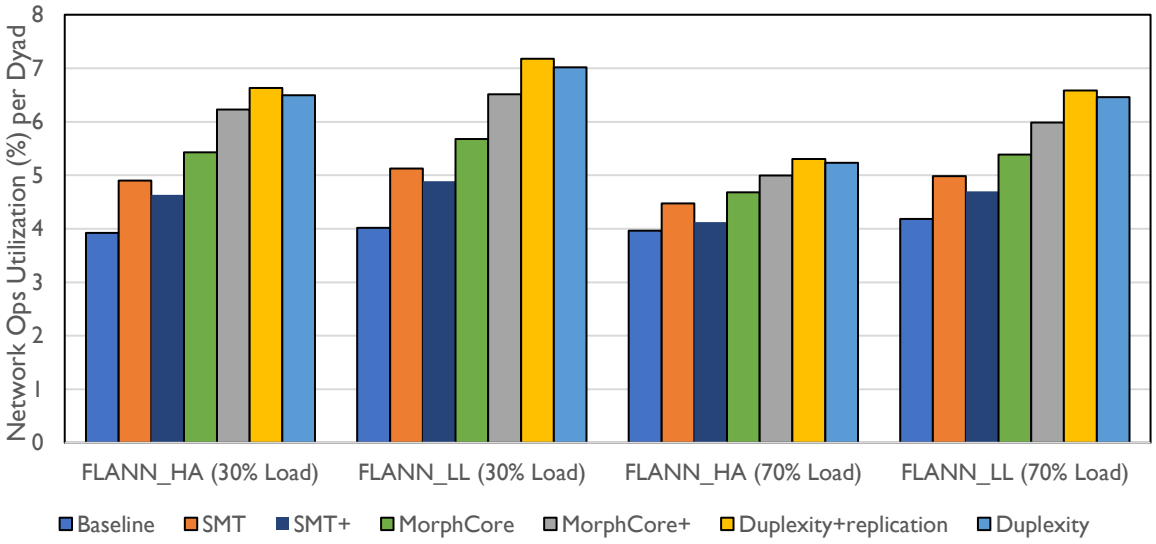


Figure 2.6: Network BW (IOPS) utilization (%) per dyad.

We consider a single FDR 4x Infiniband link to calculate network bandwidth utilization, following [15]. Most NICs impose two bandwidth constraints: a maximum data rate, and a maximum I/O operations per second (IOPS), respectively 56Gbit/s and 90M ops/s for FDR [135, 154]. As our workloads issue single–cache-line remote accesses, they are IOPS-limited. Figure 2.6 reports network IOPS utilization per dyad, which largely tracks core utilization. Duplexity improves average network utilization over the baseline and SMT by 58% and 29%, respectively. The key takeaway is that, although the main purpose of Duplexity is to improve compute utilization, it also improves utilization for other resources.

Further, Figure 2.6 confirms that Duplexity incurs requirements that fall within current networking capabilities: the maximum IOPS of each dyad is less than 7.1% of the FDR capability. Hence, 14 dyads can share one NIC port. Further scalability is possible with multiple NICs or through continued scaling of Infiniband technology to higher rates [15, 184].

2.9 Related Work

To the best of our knowledge, Duplexity is the first work to provide architectural support to fill the utilization holes caused by killer-microsecond, without sacrificying QoS and tail

latency. Concurrent to Duplexity, [30] proposes a software multithreading and prefetching-based solution to hide μs-scale memory accesses and μDPM [31] seeks to improve server energy-efficiency in the killer microseconds era. While Duplexity is a server design that targets general μs-scale I/O accesses and idle periods, the approach used in [30] is only amenable to cachable memory accesses and provides no QoS guarantees.

We review related studies on the key aspects of Duplexity.

Co-location and isolation. There is a large body of work that aims to improve processor utilization in data center workloads by co-locating batch and latency-sensitive applications using mechanisms to minimize interference among co-running applications [131, 213, 39, 122, 224]. Bubble-Flux [213] and Bubble-up [131] are online schemes that detect interference and identify "safe" co-locations to bound performance degradation while maximizing core utilization. Heracles [122] focuses on latency-critical workloads and employs isolation techniques to minimize interference between latency-critical and batch workloads. Dirigent [224] seeks to improve utilization by minimizing variation in latency-critical workloads. Finally, Paragon [38] and Quasar [39] use online classification techniques to co-locate workloads that are unlikely to interfere. These studies do not co-locate applications on different hardware threads of the same core, likely because SMT co-location (without hardware support to mitigate interference) results in drastic tail latency increases [122]. Moreover, these studies consider application behavior at millisecond (and higher) time-scales; they do not seek to address μs-scale stalls.

Further work addresses thread interference in SMT cores using specialized performance accounting hardware [52], shared resource usage tracking [25, 24], performance sampling [179], and competition heuristics [200]. However, most of the classic works do not provide QoS guarantees with respect to tail latency. Lo et al. [122] show that SMT co-location of batch and latency-critical threads can have catastrophic impacts on the tail latency and, in particular, Google's latency-critical workloads are not able to meet their QoS targets if co-located with batch threads without isolation mechanisms. SMiTe [220]

and Elfen scheduling [215] aim to minimize interference while providing QoS guarantees. SMiTe [220] follows Bubble-Up to identify co-locations that minimize QoS violations by determining workload contentiousness and interference-sensitivity. However, their results show that even the best SMT co-locations may violate QoS for $\sim$20% of requests. Elfen scheduling [215] prioritizes the latency-sensitive thread over the batch thread; the batch thread polls regularly to see if the latency-sensitive thread is running and then voluntarily deschedules itself. However, polling at μs time scales implies untenable overhead. Furthermore, both SMiTe and Elfen scheduling only consider a single batch thread, which we show is insufficient if the batch threads also incur μs-scale stalls. With many batch threads, all storage-based components of the core (especially L1 caches) become very important and it would be essential for the core to incorporate some isolation mechanism with respect to such resources, which neither SMiTe nor Elfen scheduling provides.

There have been many proposals for isolation and partitioning schemes with respect to shared last level caches [49, 166, 174, 130, 182, 211, 98, 85, 67, 78] and memory bandwidth [223, 85, 48, 144, 149, 176]. These proposals are orthogonal and can compose with Duplexity.

Reconfigurable/heterogeneous architectures. Other work aims to exploit thread-level parallelism (TLP) through microarchitectural reconfiguration. Such designs provision numerous simple compute units/cores that can either serve multiple threads individually or be ganged together to build more powerful cores when TLP is low or peak single-threaded performance is needed [204, 102, 84]. MorphCore [101] takes the opposite approach and morphs a wide OoO core into a multithreaded SMT core when threads are abundant. In conjoined/composite cores [110, 125], two cores share components to increase efficiency, as in Duplexity. Duplexity differs from these architectures as it seeks to fill killer microsecond stalls and prevent QoS harm for the latency-critical microservices.

Heterogeneous multicores, which incorporate heterogeneous cores representing different points in the power/performance design space, have been proposed to improve

energy-efficiency [109, 111]. Recent proposals allocate threads/tasks at either service-level [158] or request-level [71] to particular core types to improve throughput, density, and energy-efficiency, while meeting QoS targets. Duplexity fits within this class of multicore architecture, introducing the novel notions of dyads and thread borrowing.

2.10 Conclusion

Duplexity is a server architecture that aims to maximize performance density and energy efficiency by filling the *killer microsecond* utilization "holes" of microservices. These holes result from stalls due to accessing fast I/O devices or brief idle periods between requests. Neither existing microarchitectual techniques nor OS context switches can hide μs-scale stalls. Duplexity couples a latency-oriented master-core and throughput-oriented lender-core into a dyad. The master-core primarily executes a latency-critical master-thread. However, when idle or stalled, the master-core morphs into a multithreaded throughput mode and borrows filler-threads from the lender-core to fill utilization holes. By provisioning separate memory paths for the master and filler-threads, Duplexity protects master-thread cache state facilitating fast restart when a stall resolves. Our evaluation shows that Duplexity improves core utilization and iso-throughput tail-latency by 1.9$\times$ and 2.7$\times$ over an SMT-based server design.

CHAPTER III

The Queuing-First Approach for Tail Management of Interactive Microservices

3.1 Introduction

Online Data Intensive (OLDI) services (e.g., web search) traverse terabytes of data with strict latency targets [11]. Managing high-percentile tail latencies is a key problem in designing such services. First, to guarantee user satisfaction, services must meet strict response time Service-Level Objectives (SLOs), especially for tail latencies [94, 41]. Second, such services typically communicate via fan-out patterns wherein datasets are "sharded" across numerous "leaf" servers and their responses are aggregated before responding to the user. As such, overall latency is often dictated by the slowest leaves (i.e., the "tail at scale" effect [37]).

High tail latencies arise from two effects. First, such applications' service time distributions include outlying requests that take much longer (10×-100× or more) than the mean [99]. Some requests may require exceptional processing time depending on their arguments (e.g., search engines [11, 70]) or query types (e.g, sets vs gets in key-value stores [82, 99]). Some requests are delayed by system interference, such as from garbage collection, page deduplication, synchronous huge-page compaction or network stack impediments [96, 37]. In other cases, scheduler inefficiencies, power state transitions, suboptimal interrupt routing,

poor NUMA node allocation, or virtualization effects may contribute to long tail latencies [115]. Finally, interference from co-located workloads can cause slowdown due to contention for shared caches, memory bandwidth, or global resources like network cards or switches [122, 138].

A second key contributor to applications' end-to-end latency distribution are queuing effects [41]. Queuing arises at numerous layers causing some requests to wait for others [37]. Whereas queuing also affects average performance, its effect on tail latency may be catastrophic. To achieve performance stability, systems must be engineered such that the overall request arrival rate is lower than the aggregate system capacity (service rate). However, as both rates fluctuate, arrivals may temporarily outstrip service capacity, causing requests to queue. Queueing delay is most apparent under high system load. However, in this chapter, we make the case that queuing effects drastically magnify the impact of rare system events/hiccups and can result in high tail latencies even under modest load. Due to head-of-line (HoL) blocking, many requests are delayed by an exceptionally slow one that stalls a server/core; these delayed requests account for a bulk of the latency distribution tail.

Through stochastic queuing simulation [134], we show that improving a system's queuing behavior often yields much greater benefit than mitigating the individual system hiccups that increase service time tails. We suggest two general directions for improving system queuing behavior: *Server Pooling*, and *Common-Case Service Acceleration (CCSA)*. Server pooling is the practice of redesigning system architecture to change single-server ("scale-out") queues into multi-server ("scale-up") ones; that is, rather than enqueuing requests at distinct servers/cores, a single queue is shared among many (i.e., converting c G/G/1 queues into a G/G/c). Server pooling greatly reduces queuing delay and can completely eliminate queueing with enough servers (i.e., high enough c). Pooling smooths fluctuations in both arrivals and service, making the system behave more like one with deterministic inter-arrival and service times. Especially for high-disparity service time distributions (i.e., rare system events/hiccups), server pooling reduces the overall tail latency by breaking HoL

blocking and preventing nominal requests from waiting behind exceptionally long ones. Even a modest degree of concurrency allows many short requests to drain past stalled ones, substantially reducing weight in the latency distribution tail.

CCSA improves systems' queuing behavior by deploying optimizations that target common-case service behavior (as opposed to optimizations that target directly rare/slow requests or hiccups). It may seem counter-intuitive to improve *tail* latency by optimizing *typical-case* request performance. But, queuing delays are greatly impacted by the average load, which depends more on typical-case service time than rare cases.

In single-server systems, CCSA has little impact when the service variance is excessively high (i.e., HoL blocking is common), as nominal requests queue behind rare, slow ones regardless of how fast the nominal requests are processed. But, if there is sufficient concurrency (e.g., by using server pooling) that slow requests rarely occupy all servers, then CCSA provides enormous benefit by allowing nominal requests to drain past slow ones, drastically reducing wait time. Importantly, we show that, with concurrency, CCSA is more effective than reducing directly either the length or the probability of rare hiccups. Since finding and mitigating tail events is hard due to their myriad causes [214], we believe this observation is encouraging—we can reduce tail latency without engaging in "whack-a-mole" with rare system hiccups.

In short, we argue that cloud system designers should invest optimization effort first into (1) reducing HoL blocking through higher concurrency and improved queuing discipline (i.e., server pooling) and then into (2) optimizing common-case performance to improve mean service time. Both of these approaches may have greater impact and are easier to achieve than directly pinpointing and mitigating rare cases and hiccups. Whereas server pooling smooths out arrival and service variability, CCSA reduces the effective system load. The relative impact of the two approaches depends critically on the system load and service time variance. CCSA's effectiveness improves as service times become more normal and/or concurrency increases. We build a simple regression model on concurrency and service

time variance to estimate HoL blocking and indicate whether server pooling or CCSA is more beneficial in reducing tail latency. System designers can use this model to guide optimization effort and estimate its impact.

3.2 Background and Methodology

Most interactive cloud services can be modeled as $A/S/c$ queuing systems (based on Kendall's notation [72]), where A specifies the request inter-arrival time distribution, S the service time distribution, and c the number of concurrent servers. Regardless of the distributions, the average arrival rate (λ) must be lower than the average aggregate service rate of all servers (μc, with μ as the average service rate of a single server); otherwise, requests queue without bound.

The most common queuing models used in analytical studies are $M/M/c$ systems*, where both inter-arrival and service times follow exponential distributions. It can be shown that the exponential distribution is the only continuous distribution with the memoryless property (i.e., occurrence of events is independent of the system's history) [72]. An interesting property of exponentially distributed random variables is the constant ratio between their mean values and all of their quantiles (including the median and all-percentile tails), as shown in Equation (1). Due to the memoryless property of exponential distributions, $M/M/c$ queuing systems can be easily analyzed with Continues Time Markov Chains (CTMCs) and have closed-form solutions for many of their parameters, such as average waiting and sojourn (waiting plus service) times.

$$P(S > \alpha E(S)) = e^{-\alpha} \tag{1}$$

Neither inter-arrival nor service times of interactive cloud services are perfectly modeled by exponential distributions. But, since requests usually originate from a large pool of

*M stands for Markovian.

independent sources (e.g., many distinct users), they typically mimic Poisson (memoryless) arrivals; prior studies have observed that inter-arrival time distributions usually have small coefficients of variation (mostly, between 1 and 2 [134]). As such, inter-arrival processes can be well approximated with an exponential distribution ($CV = 1$) with little fidelity loss [133]. Service time distributions, in contrast, may have long tails; some requests encounter rare hiccups that increase service time by 10×-100× (or even more) over the mean—much larger than the ratio of the 99^{th} percentile and mean values in the exponentially distributed services times of $M/M/c$ systems (~ 4.6, based on Equation (1)). Hence, interactive cloud services are often investigated using $M/G/c$ queuing models[†] [132, 99].

Unfortunately, $M/G/c$ queuing models do not have closed-form solutions for average waiting/sojourn times and the accuracy of existing approximations, which use only a few moments, is poor [68]. Furthermore, to the best of our knowledge, there is no widely-used approximation for waiting/sojourn time quantiles of these systems. Thus, we use stochastic queuing simulation, based on the BigHouse methodology [134], to measure the tail latency of such $M/G/c$ systems. We simulate the queuing system until we achieve 95% confidence intervals of 5% error in reported results. We consider the First-Come-First-Served (FCFS) queuing discipline as prior work [115, 207] shows it to be the best non-preemptive scheduling policy when tail latency is the metric of interest.

We model "nominal" request performance by drawing service times from an exponential distribution with mean $1/\mu_n$. Then, to represent rare/slow requests, which we call "hiccups", with probability p_h we add an additional delay drawn from a second exponential distribution with mean $1/\mu_h$. We vary both p_h and the ratio of μ_n/μ_h in our experiments. This hybrid model is similar to the dual-branch Hyperexponential distribution, which is widely used as a phase-type distribution for approximating heavy-tailed systems [72]. We study analytical distributions as they are easier to understand and their parameters can be tuned to model various real scenarios.

[†] G stands for General.

The intent of our approach is to model the near-memoryless nominal behavior of cloud services and then overlay an independent distribution to model hiccups. We consider hiccups that are (1) $10\times$ longer than average, affecting 1% of requests, and (2) $100\times$ longer affecting 0.1% of requests. (1) represents unusual code paths that arise in e.g., web search. As an example, Microsoft observes a bimodal distribution for Bing search [70], wherein most requests incur latencies close to the mean but occasional requests require an order of magnitude more processing time due to their complicated search queries. They report a $27\times$ ratio between the 99^{th} percentile tail and the median latency (which is usually smaller than the mean). Similarly, Google reports a 1ms median leaf service time with 99^{th} percentile tail latency of 10ms [37]. (2) represents rare pauses that arise due to system activities and interference. As an example, [202] studies a multi-tier web application and identifies a similar bimodal distribution incorporating rare requests with less than 1% probabilities that take $30-40\times$ longer than the mean due to "transient events", such as JVM garbage collection or voltage/frequency state transitions.

3.3 The Queuing-First Approach

Requests may incur an end-to-end latency in a high percentile tail either because the request itself incurred a rare hiccup or due to queuing delays. Queuing greatly magnifies the impact of few, rare hiccups by causing nominal requests to queue behind one with a hiccup and incur high sojourn times. With deterministic or memory-less service times, queuing arises primarily due to request bursts, wherein the instantaneous arrival rate exceeds the average service rate. However, with high-disparity service time distributions, queuing is caused mostly by HoL blocking, wherein the instantaneous service rate drops temporarily well below the average request arrival rate.

The differing nature of queuing has important implications. First, with high-disparity service, queuing can arise even at low load; when a slow request stalls the server for a long time, many requests may queue behind it, even if the arrival rate is low. Second, it increases

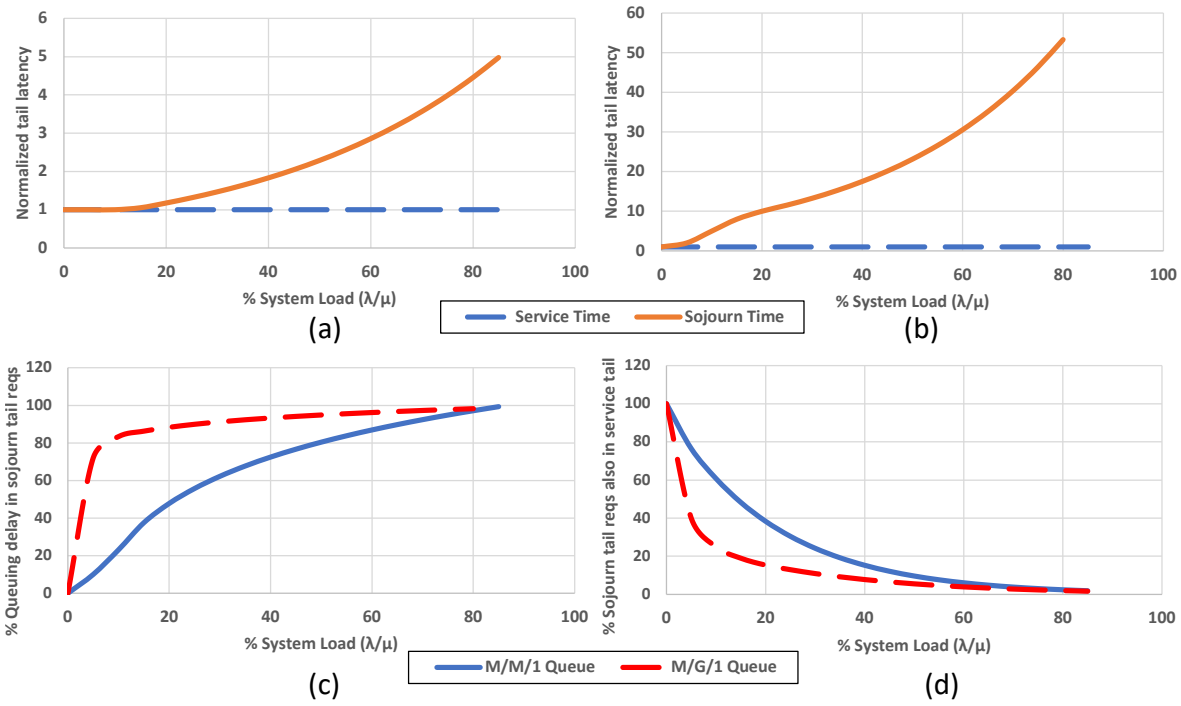


Figure 3.1: (a) Normalized service- and sojourn-time 99^{th} percentile tail in an M/M/1 queue, (b) normalized service- and sojourn-time 99^{th} percentile tail in an M/G/1 queue, (c) average % wait time in sojourn-time tail requests, and (d) % of sojourn-time tail requests that are also in the service-time tail. The M/G/1 queue has an exponential service time distribution but incorporates $100\times$ hiccups that occur in 0.1% of the requests.

the contribution of nominal requests to the sojourn-time tail; while hiccups directly impact few requests, such requests account for a large fraction of server utilization. As such, a substantial fraction of nominal requests queue behind the exceptional ones. As an example, in an $M/G/1$ queue where 0.1% of requests incur a $100\times$ higher-than-nominal service time, the exceptional requests account for $\sim 10\%$ server utilization. As a consequence of Poisson arrivals, $\sim 10\%$ of requests arrive during such a slow service and may also contribute to the sojourn-time tail.

Figures 4.1(a) and (b) report the normalized 99^{th} percentile tail latency of an $M/M/1$ system and its $M/G/1$ counterpart with the high-disparity service time distribution described above across various load levels. Figure 4.1(c) reports the fraction of sojourn time spent waiting by the 1% slowest requests for both $M/M/1$ and $M/G/1$ queues. Under low loads, wait time is usually small in $M/M/1$ systems and the sojourn-time tail is nearly the same

as the service-time tail. However, queuing accounts for a significant fraction of tail latency when the service time distribution is high-disparity. Furthermore, since hiccups occur with a low probability (0.1%), they do not noticeably affect the service time 99^{th} percentile tail. However, due to HoL blocking, their impact on the sojourn-time tail is large under both low and high loads.

Figure 4.1(d) reports the percentage of requests in the sojourn-time tail that also contribute to the service-time tail. Under both low and high loads, the percentage is much higher in the $M/M/1$ system. With high-disparity service times, HoL blocking in the $M/G/1$ system comprises the bulk of the tail—most sojourn-time tail requests are nominal requests that queue behind exceptionally slow ones. Furthermore, as shown in Figure 4.1(c), while the fraction of queuing delay relative to sojourn time in tail requests is higher in the $M/G/1$ system, queuing still accounts for more than half of sojourn time even in $M/M/1$ systems for loads over $\sim 30\%$.

The takeaway is that, if a service incurs either high load or has a high-disparity service time distribution, end-to-end tail latency is dominated by queuing effects. As a result, improving system queuing behavior is typically more effective than seeking to directly mitigate system hiccups that cause heavy/long tails. Finding and mitigating system hiccups is hard. As such, we advocate pursuing optimizations that address queuing behavior instead.

3.3.1 Server Pooling

Figure 3.2 contrasts two different models to compose multiple servers. In the scale-out model, each server has a separate request queue and a dispatcher/load balancer steers incoming requests into different queues such that the request arrival rate of all servers is balanced. In the scale-up model, instead a single request queue is shared among all servers, which each fetch requests from the central request queue as they become idle. This model requires synchronization of the central request queue, but improves queuing.

It can be shown that the scale-up ($M/G/c$) organization always outperforms the scale-out

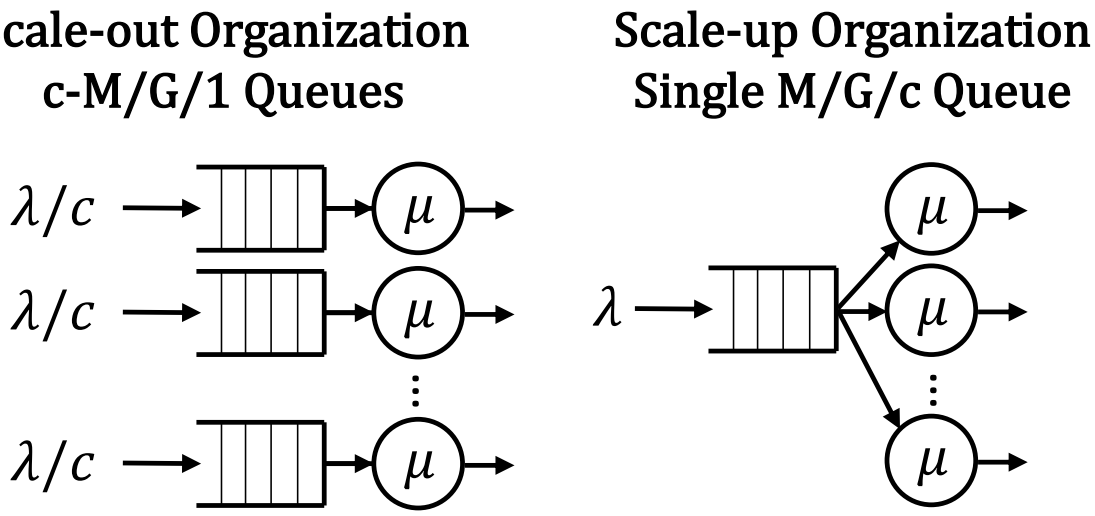


Figure 3.2: Scale-out vs. scale-up queuing organizations.

organization ($c-M/G/1$) in principle (neglecting synchronization). First, in the scale-up organization, a server will not remain idle if there are requests waiting in the central queue. However, in scale-out systems, a server may remain idle if its own queue is empty even while other servers have outstanding requests. Second, when a request takes longer than average in a scale-out organization, all the requests behind it suffer from HoL blocking delays. In contrast, in scale-up architectures, requests may be serviced by any server; stalling at one server has little impact on system-wide instantaneous service rate.

Several prior studies have observed that scale-up queuing systems outperform scale-out organizations [115, 99]. However, a large number of contemporary software systems use a scale-out queuing architecture as it is easier to implement [115]. Implementing a scale-up model across multiple machines requires remote disaggregated memory accesses or a distributed data structure, which are difficult to implement and optimize. Even within a single multi-core server, implementing a scale-up model mandates either a single synchronized data structure or a work-stealing architecture, which incur coherence traffic and are difficult to scale.

We refer to the practice of consolidating $c-M/G/1$ servers into a single $M/G/c$ system as *Server Pooling*. When service-time distributions are high-disparity, HoL blocking

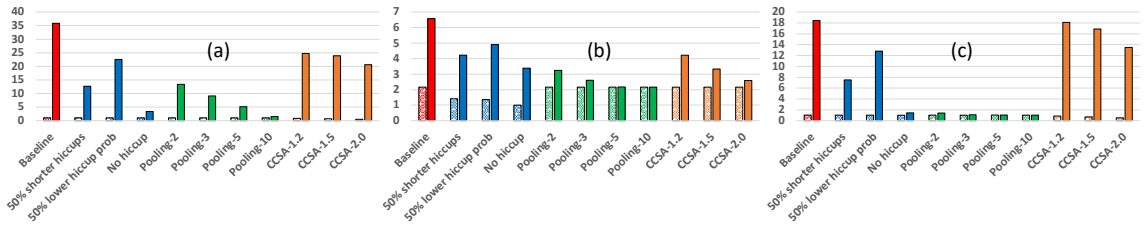


Figure 3.3: Normalized service-time (light bars) and sojourn time (dark bars) tails of an $M/G/1$ queue under different scenarios. (a) 70% load, 100× hiccups affecting 0.1% of requests, (b) 70% load, 10× hiccups affecting 1% of requests, and (c) 30% load, 100× hiccups affecting 0.1% of requests.

becomes the main source of queuing delay (and tail latency) and the gap between the two queuing organizations grows. We argue that Server Pooling can play a key role in resolving HoL blocking under such service conditions and hence should be pursued despite higher implementation complexity. In fact, Server Pooling often reduces the tail latency more than directly mitigating the rare hiccups that cause exceptionally long service.

Figure 3.3 reports the normalized service/sojourn time tail latencies in an $M/G/1$ system with different service time distributions and system loads. The leftmost red bars represent tail latencies in the presence of rare hiccups. The next group of blue bars show the tail latency where the impact (i.e., duration/probability) of hiccups has been reduced. In particular, from left to right, these bars represent cases where hiccup duration is halved, their occurrence probability is halved, and where hiccups are fully eliminated. Finally, the cluster of green bars indicate server pooling cases with varying number of servers c. (We discuss the orange bars later.)

Figure 3.3(a) considers an exponential service time distribution with hiccups that occur 0.1% of the time and last 100× longer than the average service time under 70% system load. We make three observations: First, reducing the hiccup probability is considerably less effective at reducing the overall tail than reducing their duration. The intuition is that longer hiccups cause more requests to queue and hence exacerbate tails more than shorter but more frequent hiccups. Second, pooling only two servers reduces tail latency almost as much as halving hiccup durations. Whereas it may be challenging to implement high-concurrency

data structures to enable a high degree of server pooling, sharing queues across just pairs of machines or cores is likely easier than finding and mitigating hiccups. Finally, with greater degrees of server pooling, queuing delay vanishes and the sojourn-time tail and service-time tail match. In such a scenario, end-to-end tail latency is even lower than in a system with no hiccups but without pooling.

Figure 3.3(b) reports the same results for hiccups $10\times$ longer than the average occurring in 1% of requests. Whereas the general trend matches Figure 3.3(a), the gap between the service- and sojourn-time tails is noticeably smaller even though the total service time attributable to hiccups is the same ($10 \times 1\% = 100 \times 0.1\%$). As previously observed, longer hiccups introduce more severe HoL blocking and cause more nominal requests to queue behind the exceptional ones (despite lower hiccup probability). Nevertheless, in Figure 3.3(b), pooling across only two servers, despite hiccups, is enough to reduce the sojourn time tail below that of a system without server pooling and without hiccups. Figure 3.3(c) considers the same service time distribution as Figure 3.3(a) but under lower (30%) system load. Here, whereas queuing delays are typically near-negligible under low load, the high-disparity service distribution nevertheless causes HoL blocking and a significant sojourn time tail. Interestingly, the ratio between the sojourn- and service-time tails is much higher than that seen in Figure 3.3(b) due to longer hiccups and higher HoL blocking, despite lower load. Furthermore, when HoL blocking is high but system load is low, pooling across two servers completely eliminates queuing delay.

In summary, server pooling is highly effective in eliminating HoL blocking and reducing queuing delays that otherwise arise due to rare system hiccups. Although pooling across many cores/machines is often challenging, encouragingly, we show that pooling across as few as two servers is often sufficient for large tail latency reductions.

A variety of steering and scheduling techniques can enable a scale-out system to more closely approximate scale-up system behavior. Examples include smart load-balancing schemes that steer requests to queues based on wait time estimates derived from metrics

like queue occupancy, injecting replica requests to different queues and then cancelling the redundant requests [37], and various work-stealing approaches that migrate tasks between queues [116]. While these techniques typically fall short of an ideal $M/G/c$ system, they still drastically reduce wait time and HoL blocking. Further, as shown by Wierman and Zwart [207], FCFS scheduling is only best for $M/G/1$ queues if the service-time distribution is light-tailed. Otherwise, variants of *Processor Sharing* outperform FCFS in terms of tail latency. Thus, should direct implementation of server pooling prove prohibitive in a particular system, time-multiplexing machines/cores among requests may provide an alternative to address queuing due to rare hiccups.

3.3.2 Common-Case Service Acceleration

CCSA is another general approach to improve system queuing behavior. In this approach, rather than seek to mitigate the rare hiccups that cause high service times, instead, the system designer deploys optimizations that accelerate *common case* behavior. As such, while CCSA directly reduces average service time, it has little effect on the tail of the service distribution. Conventional wisdom suggests that improving average service time does not improve tail latency; indeed, some prior work suggests trading off slower average performance to reign in tails [82, 70]. However, reducing average service time increases service rate, and hence reduces server utilization. Reduced utilization in turn reduces queuing delays.

Unlike server pooling, CCSA has little impact when HoL blocking is high, as nominal requests enqueue behind long ones regardless of how fast nominal requests are processed. The rightmost set of orange bars in Figure 3.3 report service and sojourn time tails under varying degrees of CCSA (i.e., different speedups of common-case service time). We observe a large benefit in Figure 3.3(b), where hiccups are relatively short and there is little HoL blocking; accelerating the common-case service time by only 20% (without affecting its tail) reduces the sojourn time tail almost as much as reducing the average hiccup length by half. Doubling the service rate reduces the sojourn time tail below that of a system with

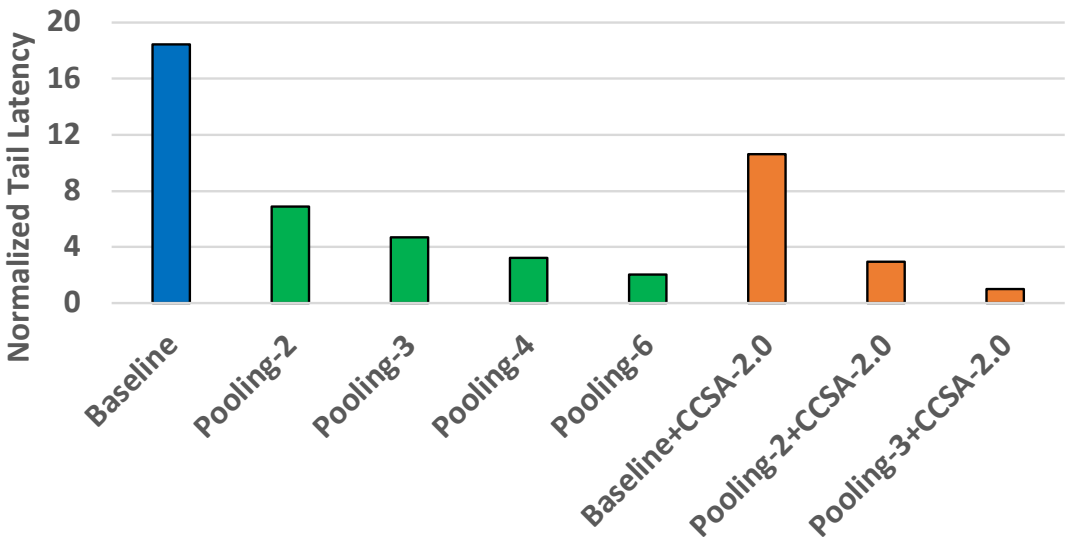


Figure 3.4: Normalized sojourn time tail latency in an $M/G/1$ queue ($100\times$ hiccups in 0.1% of requests) with various degrees of server pooling and CCSA.

no hiccups or a system with a pooling degree of two. In the remaining cases (Figures 3.3(a) and (c)), CCSA has only modest impact on the sojourn time tail, as there is more HoL blocking and insufficient concurrency for requests to avoid it.

CCSA provides greater benefit with higher concurrency (e.g., via server pooling). Even modest concurrency is sufficient to unlock CCSA's effectiveness; rare events are unlikely to occupy multiple servers at the same time, so nominal requests nearly always bypass a stalled server. As an example, Figure 3.4 considers the scenario from Figure 3.3(a) but in $M/G/2$ and $M/G/3$ systems (i.e., with server pooling). Not only is CCSA better than mitigating hiccups, it is also better than further increasing server pooling. We expect that CCSA will typically be easier to implement than hunting down and optimizing the underlying causes of rare performance hiccups as software developers are already incentivized to make the common case fast. Note that this approach is most beneficial for service distributions wherein, despite heavy/long tails, most of the system utilization arises from nominal requests. For example, in our modeled distributions, $\sim 10\%$ of system utilization is spent on hiccups. However, in many power-law distributions, tail events contribute to $80-90\%$ of

the distribution; CCSA would not be as effective in such scenarios.

3.3.3 Discussion

We expect CCSA to be more beneficial than server pooling, as CCSA reduces the effective system load, while server pooling has no effect on load/utilization. However, as we showed in the previous subsection, CCSA is only effective in the absence of sever HoL blocking. When HoL blocking is frequent (e.g., service time variance is high), CCSA no longer provides benefit as nominal requests queue behind exceptionally long ones. In such scenarios, additional concurrency must be introduced to unleash CCSA's efficacy.

In single-server systems, service time variability is a good measure of HoL blocking. For example, in Figure 3.3(a), where $CV_{service} = 4.2$, CCSA has negligible impact; nominal requests wait behind slow ones. In contrast, in Figure 3.3(b) where $CV_{service} = 1.6$, (near the $CV_{service} = 1.0$ of $M/M/*$ queues), CCSA is more effective than server pooling. However, $CV_{service}$ only reflects HoL blocking in single-server systems. We suggest the inter-departure time variability of a saturated queue (when queuing probability is close to 1.0) to measure HoL blocking in multi-server queues. In saturated single-server queues, the inter-departure time distribution is the service time distribution ($CV_{service} = CV_{departure}$). However, with multiple servers, departures interleave, reducing inter-departure time variability. For example, in Figure 3.4, which is similar to Figure 3.3(a) but with an additional 1-2 servers, the $CV_{departure}$ drops (from 3.0) to 1.7 and 1.1, respectively. As a result, in the $M/G/2$ case, CCSA yields almost the same benefit as pooling. In the $M/G/3$ case, where HoL blocking resembles that of an $M/M/*$ queue with $CV_{departure} = 1.0$, CCSA yields much better results than server pooling.

We find that a simple regression model can predict the $CV_{departure}$ of a saturated $M/G/c$ queue based on its $CV_{service}$ and the number of servers (c). We construct the model by simulating saturated queues with a set of heavy-tailed distributions with different $CV_{service}$ and measure their $CV_{departure}$. We observe that small degrees of server pooling quickly

reduce HoL blocking. Therefore, we postulate an exponential decay effect for the number of servers. Also, we note that $CV_{departure}$ may not decrease below 1.0 as the inter-departure process becomes near-memoryless around $CV_{departure} = 1.0$, where the ratio of tail-to-average cases does not decrease through higher concurrency (see Equation (1)). As a result, we suggest a regression model of the form of Equation (2) and tune its parameter using the Least Squares method. We find its average error to be less than 13%.

$$CV_{departure} \approx (CV_{service} - 1)e^{-0.8(c-1)} + 1 \tag{2}$$

Using this model, we can derive $CV_{departure}$ as a proxy for the HoL blocking rate and predict how it is affected by server pooling. Alternatively, cloud system architects may perform Stochastic Queueing Simulations, similar to our approach, and directly measure $CV_{departure}$ instead of predicting it. When the system approaches the $CV_{departure} = 1.0$ of $M/M/*$ queues, blocking becomes rare; the remaining tail of the sojourn time distribution is then primarily due to service time tails or high load. Under low load, queuing delays vanish with sufficient server pooling; remaining sojourn time tails reflect only service tails. Under high load, HoL blocking will no longer be the dominant source of queuing delays when sufficient concurrency has been introduced. As such, with sufficient server pooling (often just 2-3 servers), CCSA becomes more effective than further server pooling.

In short, we recommend developers follow a simple optimization sequence to address tail latency in their services: (1) introduce server pooling until HoL blocking is sufficiently mitigated; (2) if load is high, introduce CCSA; (3) if end-to-end tails remain unacceptable, only then seek to directly optimize rare, high service latencies.

3.4 Conclusion

Managing high-percentile tail latencies is key to designing user-facing cloud services. Rare system hiccups or unusual code paths make some requests take 10×-100× longer than the average. Prior work seeks to reduce tail latency by trying to address primarily root causes of slow requests. However, often the bulk of requests comprising the tail are not these rare slow-to-execute requests. Rather, due to head-of-line blocking, most of the tail comprises requests *enqueued behind* slow-to-execute requests. Under high-disparity service distributions, queuing effects drastically magnify the impact of rare system hiccups and can result in high tail latencies even under modest load. We demonstrated that improving the queuing behavior of a system often yields greater benefit than mitigating the individual system hiccups that increase service time tails. We suggested two general directions to improve system queuing behavior——*server pooling* and *common-case service acceleration*——and discuss circumstances where each is most beneficial.

CHAPTER IV

Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices

4.1 Introduction

Modern user-facing cloud services (e.g., web search, social media) must meet stringent Service Level Objectives (SLOs) to ensure responsiveness to millions of daily users [11, 183]. Often expressed in terms of (e.g., 99^{th} percentile) tail latency, SLOs target the latency of the slowest requests, and thus bound the slowest interaction a user may have with the service. The "tail at scale" effect [37] makes tail-tolerant computing even more challenging—such services typically communicate via fan-out patterns wherein datasets are "sharded" across numerous "leaf" servers and their responses are aggregated before responding to the user. As such, the end-to-end latency is often dictated by the slowest leaves.

Two effects can lead to high tail latencies. First, applications' service time distributions often include rare cases that take much longer (10×-100× or more) than the mean [99]. Such tasks may require extraordinary processing time and/or trigger unusual code paths [70, 82, 91]. In other cases, system effects, such as from garbage collection [202, 37], memory management activities [177, 156], virtualization [212], network stack impediments [96, 115, 203], or co-runner application interference [220, 122, 38] may delay tasks.

Queuing effects are a second key contributor to end-to-end tail latency [41]. Queuing,

where some requests must wait for others, arises at many system layers [37, 175]. Whereas queuing can affect average performance, its effect on tail latency may be devastating. For stable performance, systems must be engineered to ensure overall request arrival rate is below the aggregate system capacity (service rate). However, as both rates fluctuate, arrivals may briefly outstrip service capacity, causing requests to queue. Queuing delay is most apparent under high service time variability and/or high system load. Under high-disparity service distributions, many requests become delayed by an exceptionally slow one that stalls a server/core—a phenomenon called Head-of-Line (HoL) blocking. These delayed requests account for the bulk of the latency distribution tail under moderate-to-high loads [139].

In this chapter, we introduce *Q-Zilla* as an algorithmic framework to tackle the problem of tail latency from a queuing perspective. In Q-Zilla, we make two distinct contributions: First, we propose *Server-Queue Decoupled Size-Interval Task Assignment (SQD-SITA)* as an efficient scheduling algorithm for high-disparity service distributions to minimize tail latency. SQD-SITA is inspired by an earlier algorithm, SITA [35, 73], which seeks explicitly to address HoL blocking by providing an "express-lane" for short tasks, protecting them from queuing behind rare, long ones. However, SITA requires prior knowledge of tasks lengths to steer them into their corresponding lane—an impractical assumption. Furthermore, whereas SITA is generally effective at reducing queuing delay and tail latency, it can fall short of the performance of a single-queue $M/G/k$* system when some lanes become underutilized. To overcome these challenges, SQD-SITA uses incremental preemption to avoid the need for a priori task-size information, and dynamically reallocates servers to lanes to boost server utilization. SQD-SITA never falls short of $M/G/k$ performance. We further introduce an enhanced variant of SQD-SITA, called Interruptible SQD-SITA (ISQD-SITA), which maximizes server utilization and further improves tail latency at the cost of additional preemptions.

*Kendall's Notation: A/S/k [A/S: arrival/service distribution, k: number of servers, M: Markovian/memoryless (exponential) distribution, G: general distribution]—because requests usually originate from many independent sources (e.g., distinct users), they typically mimic Poisson (memoryless) arrivals [134, 72].

Second, as an example realization of the Q-Zilla framework, we propose *CoreZilla*, a microarchitecture to minimize the tail latency of μs-scale microservices. Modern internet services use distributed microservice architectures, wherein a complex application is decomposed into numerous discrete microservices that interact over high-performance data center networks using remote procedure calls (RPCs) [187, 185, 58]. Many cloud service companies, including Amazon [188], LinkedIn [189], Netflix [193], and SoundCloud [159] have adopted microservice-based architectures. Example microservices include content caching [56, 54], protocol routing [118, 150], key-value lookup [92, 142], query rewriting [8], or other steps performed across various application tiers [59].

Managing tail latency is inherently more difficult for microservices, as individual RPCs/-tasks are often only a few microseconds [138, 9]. Due to these short task lengths, it is often prohibitive to implement a "scale-up" queuing organization, wherein a single task queue is shared among all cores, as this organization leads to high contention on the shared queue—all cores must synchronize frequently to retrieve new tasks and the excessive synchronization costs may outweigh the benefits of sharing the task queue across cores. Nonetheless, such systems can adopt a hierarchical queuing scheme, wherein each core maintains a distinct queue that is shared only among hardware threads running on that core, achieving strong cache affinity for the local task queue.

Our proposal, CoreZilla, implements a hierarchical scheduling algorithm across hardware contexts in a Simultaneous Multithreading (SMT) core. It incorporates an automatic load adaptation scheme that dynamically tunes the number of physical contexts and schedules virtual contexts on them using ISQD-SITA. CoreZilla minimizes queuing delay and tail latency at each core, obviating the need for a cross-core scale-up queuing architecture and its associated synchronization and cache coherence overheads. Our evaluation demonstrates that CoreZilla improves tail latency over a conventional SMT core by 2.25×, 3.23×, 4.38× with 2, 4, 8 contexts, on average, respectively. We further compare CoreZilla to a hypothetical 32-core scale-up system with idealized (zero-overhead) synchronization. CoreZilla with

8 contexts still outperforms the idealized scale-up design by 12%, due to superior task scheduling.

4.2 Background and Motivation

4.2.1 Queuing Organizations

Prior work has considered two different approaches to compose multiple servers: scale-out and scale-up [139]. In the scale-out model ($k-M/G/1$), a dispatcher balances incoming tasks among separate request queues dedicated to each server. In the scale-up model, servers instead fetch tasks from a single, shared queue. In principle, in terms of average and tail response time, the scale-up ($M/G/k$) organization always outperforms the scale-out organization. In the scale-up organization, no server will idle if there are tasks waiting in the central queue. However, in scale-out systems, a server with an empty queue will remain idle even while others have outstanding tasks. Furthermore, for scale-out systems, when a task takes longer than average all the tasks behind it suffer from HoL blocking. In contrast, in the scale-up model, tasks may be serviced by any server; stalling at one server has less impact on the system-wide instantaneous service rate.

Figure 4.1 reports the normalized 99^{th} percentile tail latency of five different microservices measured at 70% load, on a Xeon processor with 16 cores and Hyperthreading. More details about the microservices are presented in Section 4.7. Different bars report the tail latency under (1) a scale-out queuing organization, wherein each core has a distinct task queue, (2) a hierarchical queuing organization, wherein the two hyperthreads of each core share a single task queue, (3) a scale-up organization, wherein a single task queue is shared among all cores, and (4) a theoretical scale-up model, wherein the costs of the synchronization and cache coherence are neglected (not measured on real hardware).

As shown in Figure 4.1, whereas a scale-up organization can theoretically result in 8.3× lower tail latency, on average, compared to a scale-out organization, it can only reduce

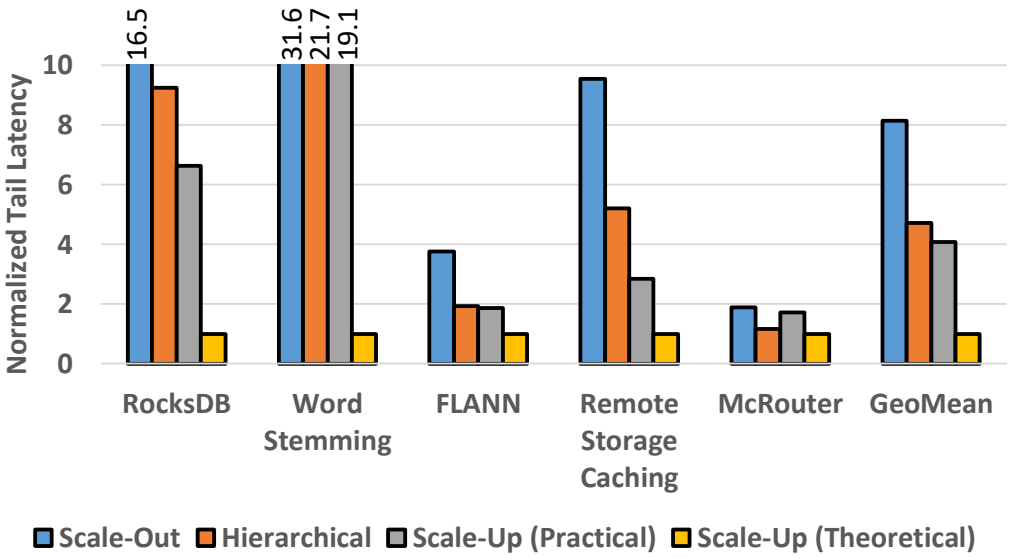


Figure 4.1: Normalized 99^{th} percentile tail latency of different queuing organizations (16 dual-threaded cores).

the tail latency by 1.93× in practice due to communication and synchronization costs of the shared queue. However, a hierarchical approach only achieves 16% higher tail latency than a practical scale-up organization because (1) sharing the queue across hyperthreads within a core minimizes the synchronization/coherence overheads [63], and (2) as observed in prior work [139], only a small degree of concurrency is sufficient to eliminate the HoL blocking and allow the nominal tasks to drain past the rare, long ones. Interestingly, for microservices like McRouter, which do not exhibit heavy-tailed service distributions, the hierarchical approach results in lower tail latency than a practical scale-up organization.

Nonetheless, there is a still a $\sim 4\times$ gap between the hierarchical approach and the theoretical scale-up organization (with no synchronization or cache coherence overheads). Our goal is to design scheduling algorithms that can be implemented across hardware threads within a single physical core, to bridge the gap between theoretical and practical (hierarchical) queueing schemes.

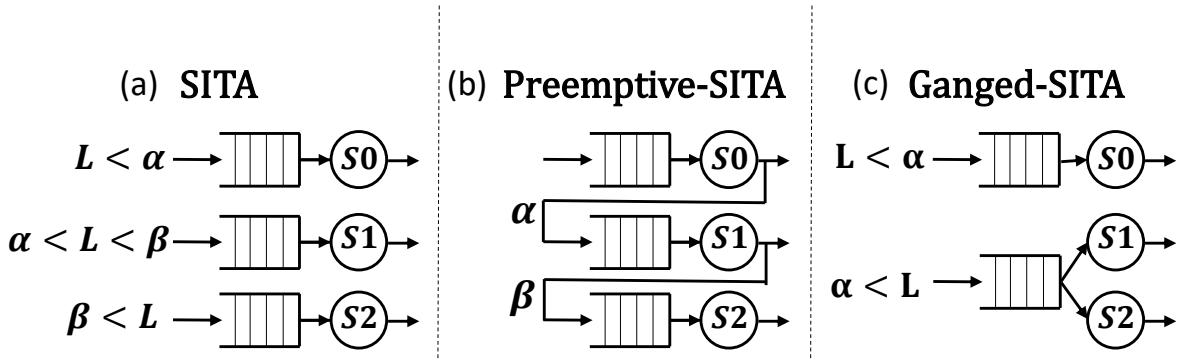


Figure 4.2: (a) Size-Interval Task Assignment (SITA); (b) SITA with incremental preemption (Preemptive-SITA); and (c) SITA with Server Ganging (Ganged-SITA). α and β represent cutoff points. L refers to task lengths.

4.2.2 SITA Scheduling

In an $M/G/k$ system with high service-time variability, especially with moderate-to-high load, it is probable that all servers become occupied by long tasks. In these cases, short tasks become enqueued behind long ones and suffer substantially from HoL blocking, increasing tail latency. The Size-Interval Task-Assignment (SITA) [35, 73] scheduling policy explicitly addresses this problem by providing an "express-lane" for the short tasks, protecting them from rare, long ones. Unlike $M/G/k$, SITA considers multiple servers with dedicated queues for each, similar to scale-out ($k-M/G/1$) systems. However, in contrast to scale-out systems, SITA assigns cutoff points to task-size intervals and steers tasks into queues based on the interval to which their size belongs. As an example, Figure 4.2(a) illustrates a SITA-scheduled system with three servers and cutoff points α and β. By providing an express-lane for tasks that are shorter than α, SITA prevents them from being enqueued behind long ones, to reduce tail latency under high service-time variability.

There are various approaches for tuning SITA cutoff points, such as equalizing the load across all servers [73]. However, to minimize the end-to-end response time, cutoff points must often be set in a way that intentionally unbalances load to favor either short or long tasks [35]. Finding the minimal cutoff points for SITA is, to date, an open problem and is usually done via empirical search, especially if the number of servers is small [72].

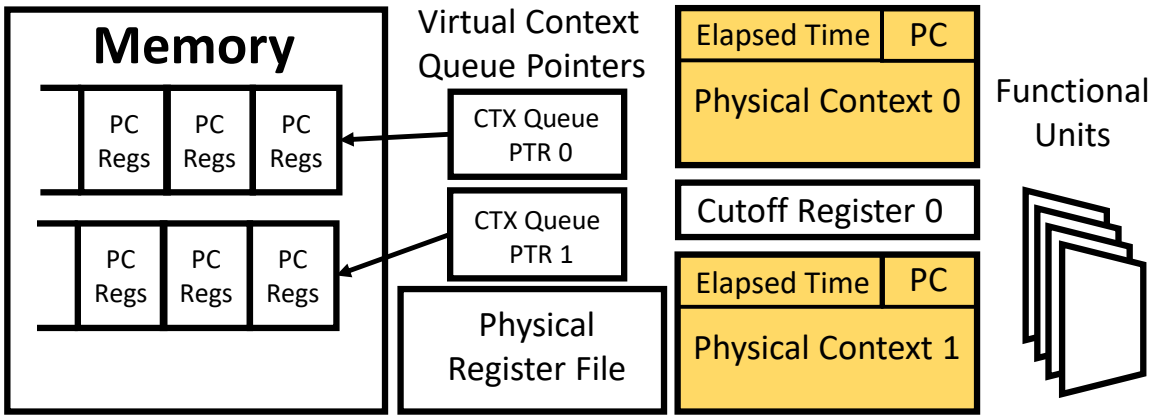


Figure 4.3: A two-way Express-Lane SMT (ESMT) core.

Although SITA is often effective at reducing tail latency and queuing delay under most high-variability service distributions, it suffers from two shortcomings: First, whereas $M/G/k$ fully utilizes all servers, SITA fails to do so, as it pre-assigns tasks to servers while ignoring their load, similar to scale-out systems; while there might be outstanding tasks at one queue, other servers may be idle waiting for new tasks to arrive. Second, SITA requires task sizes to be known in advance, which is an impractical assumption. We propose SQD-SITA to address these shortcomings.

4.3 Express-Lane SMT

We first introduce the Express-lane SMT (ESMT) [140] microarchitecture, as a basline for CoreZilla. An ESMT core shown in Figure 4.3, comprises two physical contexts and a fixed number (e.g., 32) of virtual contexts, organized in two context queues in dedicated memory. The ESMT core may fetch and issue instructions only from the two physical contexts. Virtual contexts must be swapped into a physical context before they may execute. The ESMT core datapath resembles an SMT core with two hardware threads, and has similar area costs and clock frequency. Similar to existing SMT cores, the physical register file holds the architectural register values of all physical contexts and additional registers that enable register renaming and out-of-order execution.

When the first physical context reaches a preset service cutoff, it stops fetching/dispatching new instructions and drains the in-flight instructions from the re-order buffer. Once all in-flight instructions are drained, the context only contains architectural registers (all temporary physical registers are released). Then, its architectural state (program counter, registers, etc.) is swapped by the virtual context at the head of the first context queue and the preempted virtual context is placed at the end of the second context queue.

Swapping the virtual contexts into and out of the core is performed via microcode operations using the *Firmware Context Switching* (FCS) mechanism [197, 138]. FCS behaves as an additional instruction sequence for swapping threads, much like that done in software by the operating system, and therefore does not impose any additional requirements on the number of register file ports; microcode r-save/r-restore operations access the register file like typical load/store instructions. However, because FCS does not incur user/kernel mode transitions or switch address spaces, it is considerably faster than software context switches; while software context switches require 5-20μs [114, 195], a typical FCS can usually be performed within only 300ns [197]. Nonetheless, each virtual context is slowed down by at least a microsecond when it is preempted and moved to the second context queue due to indirect caching effects (i.e., cold misses when the task is resumed) [138]. This is not a significant problem in ESMT as the number of preemptions per virtual context is at most one.

ESMT allocates an idle virtual context to each incoming task. By having two context queues, ESMT provides an "express lane" for nominal tasks, protecting them from HoL blocking behind rare, long ones, thereby reducing queuing delay and tail latency—ESMT implements a preemptive variant of the SITA scheduling policy (which we will formally define in Section 4.4) with only two lanes.

A major drawback of ESMT, and SITA in general, is underutilization of physical contexts (servers in SITA) as they are statically mapped to queues. As we will show in Section 4.8, ESMT bridges the small gap between a hierarchical queuing scheme and a

practical implementation of a cross-core scale-up queuing organization. However, due to said underutilization, it still significantly falls short of a theoretical scale-up queuing organization. Our goal is to design scheduling mechanisms that can be used in ESMT-like core microarchitectures to avoid underutilization while minimizing preemptions, closing the gap between hierarchical designs and theoretical scale-up organizations.

4.4 Server-Queue Decoupled SITA

We propose Server-Queue Decoupled (SQD)-SITA—a preemption-based variant of SITA that improves server utilization by dynamically reallocating servers to queues, which prevents servers from idling while tasks wait in another queue. We also introduce an enhanced variant of SQD-SITA, called Interruptible SQD-SITA (ISQD-SITA), which maximizes server utilization and further improves tail latency but may result in additional preemptions. Note that the (I)SQD-SITA scheduling algorithm may be implemented on different substrates, including multicore processors. However, in this chapter, we propose an ESMT-based implementation of these algorithms, called CoreZilla, which is well-suited for modern μs-scale microservices. We construct the SQD-SITA and ISQD-SITA scheduling algorithms in three steps:

4.4.1 Adding Preemption and Ganging to SITA

We begin our development of SQD-SITA by enhancing SITA with *incremental preemption* and *server ganging*.

Incremental preemption. Whereas SITA statically assigns tasks to lanes based on their length, SQD-SITA incrementally preempts and migrates them to the end of the next queue as they reach a pre-determined service time cut-off. We call a SITA variant that also performs incremental preemption *preemptive-SITA*, as shown in Figure 4.2(b), and compare against it in our evaluation. The incremental preemption approach of preemptive-SITA is similar to the approaches used in ESMT [140] and some software frameworks [70, 116]. Unlike SITA,

preemptive-SITA (and SQD-SITA) does not require prior knowledge of task lengths.

Server ganging. Server ganging (also called server pooling) is the practice of merging multiple scale-out queues into a single scale-up one by allowing multiple servers to share a single queue [139]. The original SITA algorithm was designed for task allocation in data center clusters, where a "server" represents a physical machine [35]. In such deployments, each server is associated with a distinct queue. However, SQD-SITA is intended primarily for scheduling tasks on cores/threads within a single machine. Therefore, it is possible to consolidate multiple queues and have fewer queues (and hence, cutoffs) than servers. Our key observation, which we will quantitatively explain in Section 4.8, is that only a few cutoffs are typically sufficient for SITA to achieve optimal isolation of long and short tasks; having a distinct cutoff per server leads to unnecessary load-imbalance, increasing queuing delay and tail latency. As a result, to construct the SQD-SITA algorithm, we start from a (preemptive) SITA variant where the number of queues is less than or equal to the number of servers, allowing a queue to be serviced by more than one server, as the first step towards server-queue decoupling. We call this variant Ganged-SITA, as shown in Figure 4.2(c), and compare against it in our evaluation.

4.4.2 Server-Queue Decoupling

The key feature of SQD-SITA is that it dynamically reallocates servers to queues to improve utilization and tail latency. We start from a preemptive-SITA system with server ganging as a strawman and look for scenarios where changing the assignment of servers to queues improves utilization without impacting performance. To this end, we derive upper and lower bounds on the number of servers that can be assigned to service tasks from each queue, and an algorithm to assign servers to queues in a way such that these bounds are met.

All tasks enter the system at queue 0 and, when they reach the predetermined service cutoff, are preempted and enqueued in queue 1, and so on. Thus, each successive queue contains longer tasks. Whereas in a conventional queuing system, servers are assigned to

particular queues, in SQD-SITA, we conceptually associate lanes with each queue, and servers join lanes to accept tasks from a particular queue. A server is only preempted when its task finishes or reaches the cutoff point where it should advance to the next lane (i.e., task preemptions are only triggered by timers, not any external events, such as new task arrivals); we will relax this assumption later when we discuss ISQD-SITA. When a task is immediately reassigned to the server it was running on before being preempted, the preemption is elided.

Reservations and starvation. In SQD-SITA, we conceive a system with N servers and M queues/lanes, numbered 0 to N-1, and 0 to M-1, respectively, where $M <= N$. We associate each lane with a positive number of *reservations*, where the sum of the number of reservations in all M lanes is equal to N. Thus, for example, if the number of servers and lanes is equal ($M = N$), each lane has only one reservation. The number of reservations specifies the minimum number of servers that must be available to serve tasks in a lane. We say that a lane *starves* if it has fewer tasks in service than its reservations while tasks wait in its queue. SQD-SITA's goal is to maximize server utilization while avoiding starvation. That is, we allow a lane to be assigned more servers than its reservations only if we can guarantee no other lane will starve.

Upper-bound criterion. To ensure a lane is assigned servers beyond its reservation only if we provably avoid starvation, we define an *upper bound criterion* to limit the maximum number of servers that may be assigned to each lane. The upper bound criterion assures that, if new tasks arrive, servers will be available at lower-numbered lanes so those lanes do not starve. The upper-bound criterion is expressed in Equation 1. For any given lane, the number of servers that may be allocated to this and all higher-numbered lanes, in total, is at most the cumulative reservations of these lanes. At a high level, Equation 1 ensures that, for any k, lanes 0 to k can always accommodate, in total, at least as many tasks as their cumulative reservations. To understand the purpose of this criterion, consider an extreme case where all the lanes 0 to k are empty. Even so, a burst of tasks might arrive and quickly flow into these lanes before existing (long) tasks running in higher-numbered lanes finish,

starving low-numbered lanes. In the special case where the number of lanes and servers are equal (i.e., each lane has a single reservation), Equation 1 simplifies to Equation 2. In this case, only one server may service lane N-1 (longest tasks), at most two in lanes N-2 or higher, 3 in lanes N-3 or higher, and so on.

$$\forall\ 0 \leq m \leq M-1\ :\ \sum_{i=m}^{M-1} servers(lane_i) \leq \sum_{i=m}^{M-1} reservations(lane_i) \tag{1}$$

$$\forall\ 0 \leq m \leq M-1\ :\ \sum_{i=m}^{M-1} servers(lane_i) \leq M-m \quad (M=N) \tag{2}$$

Lower-bound criterion. Whereas the upper-bound criterion is necessary to avoid starvation, it is not sufficient. We must also introduce a *lower-bound criterion* on the allocation of servers to lanes, denoted in Equation 3. At a high level, the lower-bound criterion ensures that a lane will receive at least as many servers as the sum of its reservation and the unused reservations of higher-numbered lanes. The key intuition underlying this criterion is that tasks that reach a cutoff and must advance to the next lane *can take their server with them* to satisfy the next lane's reservation but this must not cause their previous lane to starve. The lower bound criterion ensures this does not happen. We will provide an example later to illustrate this case.

$$\forall\ 0 \leq m \leq M-1\ :\ servers(lane_m) \geq \tag{3}$$

$$min(tasks(lane_m), \sum_{i=m}^{M-1} reservations(lane_i) - \sum_{i=m+1}^{M-1} servers(lane_i))$$

Interestingly, we show that, to maximize utilization (while ensuring no lane starves), each lane must be allocated exactly as many servers as specified by the lower-bound criterion: the upper bound criterion (Equation 1) can also be written as Equation 4 by deriving the maximum number of servers that may be allocated to each lane from Equation 1. Equation 4 shows that this maximum number equals the second operand of the *min()* function in Equation 3. When this second operand is the smaller, the upper- and lower-bound criteria match. Conversely, if the first is the smaller (the lane has fewer tasks than reservations), allocating additional servers to the lane would leave those servers idle. As a result, in

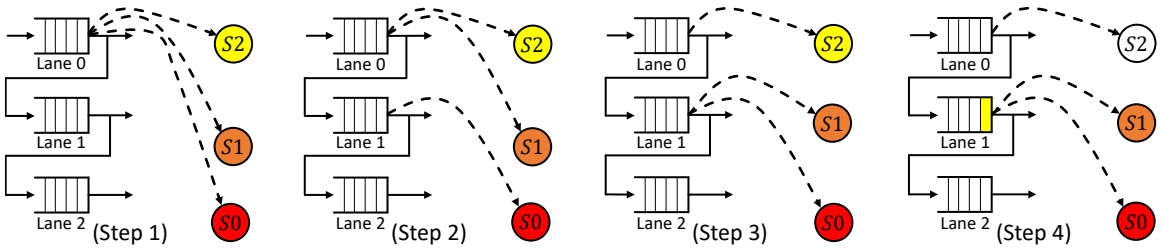


Figure 4.4: (Step 1) The initial configuration of an SQD-SITA system with three lanes and three servers, which are all initially allocated to lane 0, and (Step 2)/(Step 3)/(Step 4) when the first/second/third task reaches the first cutoff point.

SQD-SITA, each lane (except lane 0) is allocated exactly as many servers as specified by the lower bound criterion (Equation 3), maximizing server utilization and ensuring no lane starves. "Extra" servers beyond those required to satisfy the lower-bound criterion wait at lane 0 in anticipation of newly arriving tasks.

$$\forall\ 0 \leq m \leq M-1\ : \tag{4}$$

$$servers(lane_m) \leq \sum_{i=m}^{M-1} reservations(lane_i) - \sum_{i=m+1}^{M-1} servers(lane_i)$$

SQD-SITA algorithm. To ensure lanes are allocated servers to match the lower-bound criterion, SQD-SITA adopts the following algorithm: when a server becomes idle, it joins the *highest-numbered* lane where the lower-bound criterion (Equation 3) is not already met. Stated differently: when a server becomes idle, it joins the *highest-numbered* lane with a *non-empty queue*, where allocating one more server would not violate the upper-bound criterion (Equation 1 or 4).

Example. We illustrate SQD-SITA's operation in Figure 4.4. In this simple example, the number of servers and lanes are both three, and hence, each lane has a single server reservation. Initially, all servers accept tasks from lane 0. When a server becomes idle (i.e., its task finishes or reaches the cutoff point and is therefore preempted and advances to the next queue), the server joins a lane to accept a new task from the head of the corresponding queue according to the SQD-SITA procedure. Suppose three tasks (red, orange, yellow) arrive at queue 0; all three servers (S0-S2) accept tasks from lane 0 and all three tasks enter

service (Figure 4.4 - Step 1). When the red task has been serviced for a time equal to the first cutoff point, it is preempted and advances to queue 1. The newly idle server, S0, then scans the queues to seek the eldest waiting task while respecting the upper-bound criterion. In this case, S0 joins lane 1, as shown in Figure 4.4 (Step 2), and resumes servicing the task that it had previously served (the preemption is elided). Now suppose the orange task running on S1 also reaches the cutoff point; it is also preempted and migrated to lane 1. Again, the newly idle server, S1, scans the queues, finds work in lane 1 (the just-preempted orange task), and resumes serving the task (Figure 4.4 - Step 3). Note that this configuration does not the violate upper-bound criterion, which allows at most two servers to be allocated to lanes 1 and 2 in total. However, subsequently, when the yellow task also reaches the cutoff point, although the task migrates to lane 1, server S2 may not join lane 1 and resume serving the task, as the resulting lane assignment would violate the upper-bound criterion. Hence, the yellow task is preempted and appended to queue 1, yielding the state in Figure 4.4 (Step 4), wherein server S2 remains idle at lane 0 (in anticipation of new arrivals) despite the yellow task waiting at lane 1.

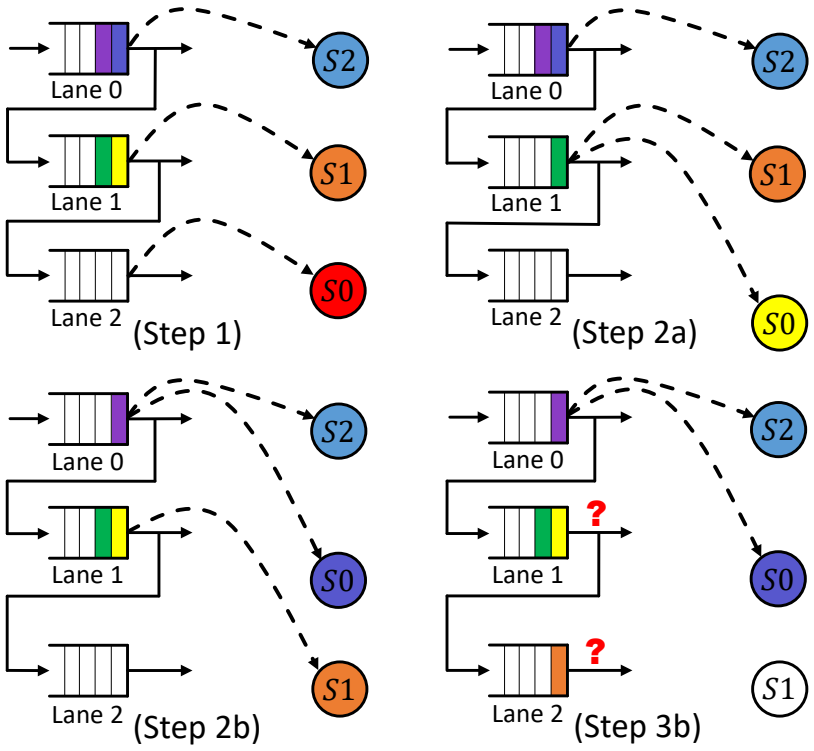


Figure 4.5: (Step 1) A valid configuration where each lane has at least one task and is allocated a server; (Step 2a) S0 follows the SQD-SITA procedure and joins lane 1 after finishing its task at lane 2; (Step 2b) S0 is allocated to lane 0 instead of lane 1 to prioritize short tasks; and (Step 3b) either lane 1 or lane 2 starves due to shortage of servers, resulted by the lower-bound violation in Step 2b.

Avoiding starvation. It may at first seem counter-intuitive that idle SQD-SITA servers scan lanes from highest to lowest to seek the eldest waiting task, as this appears to favor longer tasks over shorter ones. This policy ensures the lower-bound criterion is met at all lanes, while the the upper-bound criterion prevents too many servers from joining high-numbered lanes. We present another example to illustrate how this procedure avoids starvation. Consider the system in Figure 4.5, again with three servers and three lanes. In the scenario in Figure 4.5 (Step 1), servers S0, S1, and S2 are serving red, orange, and blue tasks and are assigned to lanes 2, 1, and 0, respectively. The indigo and violet tasks wait in the queue at lane 0 while the yellow and green tasks wait in the queue at lane 1. Lane 2's queue is empty. When S0's task completes, it scans lanes to seek the eldest waiting task, finding work (the yellow task) at lane 1 (Figure 4.5 - Step 2a). Were it instead assigned to lane 0 to prioritize shorter tasks (Figure 4.5 - Step 2b), when S1's orange task reaches the

cutoff point, is preempted, and advances to lane 2 (Figure 4.5 - Step 3b), one of the two lanes 1 and 2 has to starve as they both have tasks but there is only one server S1 available to be assigned. This example shows that greedy prioritization of short-task lanes over long-task ones can lead to starvation; the lower-bound criterion (violated in Step 2b) ensures this may not occur.

Corollaries. We note two additional provable properties of SQD-SITA: (1) no task experiences longer response time under SQD-SITA than it can experience with (server-ganged) preemptive-SITA[†], and (2) SQD-SITA minimizes preemptions as a server always follows its task when the task advances across lanes, to avoid preemption, unless doing so violates the upper-bound criterion[‡].

4.4.3 Interruptible SQD-SITA

SQD-SITA limits the number of servers assigned to each lane, according to the upper-bound criterion, to guarantee that no task experiences a longer response time than under preemptive-SITA. As a result, whereas SQD-SITA improves server utilization over both SITA and preemptive-SITA, it fails to achieve the optimal utilization, wherein servers never remain idle when tasks are queued at some lane (i.e., SQD-SITA is not work-conserving)—SQD-SITA intentionally idles servers in anticipation of new task arrivals. In this section, we propose Interruptible SQD-SITA (ISQD-SITA), which seeks to maximize server utilization by allowing servers to join lanes in violation of the upper-bound criterion *if and only if* they would otherwise remain idle.

However, to avoid starvation (assure each lane can accommodate at least as many tasks as its reservations), ISQD-SITA requires an additional preemption mechanism, which allows new arrivals to preempt running tasks (in contrast to SQD-SITA, wherein tasks are only preempted at cut-off points). When a new task arrives, if no idle server waits at lane 0,

[†]Tasks enter lanes in FIFO order and no lane ever starves.

[‡]By induction, any elder tasks are either being served, or cannot be currently served due to the upper-bound criterion (Equation 1).

ISQD-SITA scans lanes from highest to lowest to check if the upper-bound criterion has been violated in any lane. If so, it preempts the youngest running task in that lane and allocates the preempted server to the arriving task in lane 0.

ISQD-SITA algorithm. Algorithm 1 shows the high-level procedure that SQD-SITA and ISQD-SITA follow when a server is preempted. The highlighted parts are exclusive to ISQD-SITA. The procedure has two phases. In phase 1—shared between SQD-SITA and ISQD-SITA—an idle server joins the highest-numbered non-empty queue that has capacity for an additional server without violating the upper-bound criterion. Under SQD-SITA, if no such queue is found, the server will idle, waiting for new work to arrive at lane 0. Under ISQD-SITA, the server instead picks the head of the lowest-numbered non-empty queue (phase 2), in violation of the upper-bound criterion. If a new task arrives, some server must be preempted immediately to ensure the lower-bound criterion is still met. Note that even though ISQD-SITA permits upper-bound violations, it never violates the lower-bound criterion. As a result, external preemptions (i.e., non–timer-based preemptions) only occur upon new task arrivals.

Corollaries. As shown in Algorithm 1, while phase 1 scans the lanes from highest to lowest, phase 2 scans them from lowest to highest. This has two advantages: (1) it prioritizes shorter tasks, and (2) it guarantees that no tasks wait in a lane numbered lower than the one with the upper-bound violation[§]. The latter property has two useful implications: First, it retains the guarantee that no lane starves since the system behaves the same as SQD-SITA in all lanes numbered above that where the violation occurred (those lanes meet both the lower-bound and the upper-bound criteria). The yellow highlight in Algorithm 1 is an optimization that exploits this property to stop scanning lanes in phase 1 if an upper-bound violation is detected (as the rest of the lanes have empty queues). Second, (ignoring preemption cost) it ensures that no task experiences higher response time under ISQD-SITA than SQD-SITA,

[§]By induction: the first time an upper-bound violation occurs, the property holds; after that, if a task is enqueued in a lower-numbered lane than the violating lane, that task would be selected by phase 2 of the algorithm, so the property continues to hold.

Algorithm 1: SQD-SITA and ISQD-SITA pseudocodes
(Red highlight: only in ISQD-SITA — Yellow highlight: optimization)

```
1  event server S is preempted
2      server_count = 0                                                    // phase 1
3      reservation_count = 0
4      for i = M-1 to 0 do
5          server_count += lanes[i].num_servers()
6          reservation_count += lanes[i].num_reservations()
7          if lanes[i].has_waiting_tasks() and server_count < reservation_count then
8              lanes[i].allocate_server(S)
9              return
10         end
11         else if server_count > reservation_count then
12             break
13         end
14
15     end
16     for i = 0 to M-1 do                                                 // phase 2
17
18         if lanes[i].has_waiting_tasks() then
19             lanes[i].allocate_server(S)
20             return
21         end
22     end
23     lanes[0].allocate_server(S)                                         // last resort
24
25 end
26 event task arrival
27     if lanes[0].has_idle_servers() then
28         return
29     end
30     server_count = 0
31     reservation_count = 0
32     for i = M-1 to 0 do
33         server_count += lanes[i].num_servers()
34         reservation_count += lanes[i].num_reservations()
35         if server_count > reservation_count then
36             S = lanes[i].youngest_running_task()
37             preempt(S)
38             lanes[0].allocate_server(S)
39             return
40         end
41     end
42 end
43
```

again since all lanes numbered higher than the violating lane behave as in SQD-SITA and no task can be waiting in lower-numbered lanes. Therefore, if detecting new arrivals and preempting existing tasks is inexpensive, ISQD-SITA should be preferred over SQD-SITA.

4.5 Core-Zilla Microarchitecture

In this section, we describe the CoreZilla microarchitecture, which extends the ESMT design and employs ISQD-SITA to schedule tasks to hardware threads. CoreZilla enables a hierarchical queuing organization, wherein each physical core has a dedicated task queue (as in scale-out designs) to avoid the synchronization and cache coherence overheads of sharing a queue across all cores in scale-up systems. However, each core's task queue is shared among its hardware threads to minimize queuing delay and tail latency at the core, obviating the need for cross-core scale-up solutions. CoreZilla facilitates software-transparent preemptive scheduling within the core to eliminate HoL blocking and minimize tail latency. In addition to ISQD-SITA scheduling, CoreZilla dynamically tunes the number of active hardware threads, based on the system load, to yield optimal tail latency. In the following subsections, we describe the two key components of CoreZilla: *Hierarchical Scheduling* and *Automatic Load Adaptation*.

4.5.1 Hierarchical Scheduling

As shown in Figure 4.3, a strawman ESMT core is composed of two physical contexts and a fixed number of virtual contexts that are organized in two context queues in dedicated memory. CoreZilla extends ESMT to have a tunable number of physical contexts (e.g., 2-8), and virtual contexts (e.g., 32). Thus, rather than only two context queues, in CoreZilla we provision a number of context queues that can be configured to be less than or equal to the number of physical contexts. These context queues each correspond to an ISQD-SITA queue and maintain the backlog of virtual contexts ready for execution in a particular ISQD-SITA lane. Each physical context represents an ISQD-SITA server.

The CoreZilla scheduling hardware manages the assignment of physical contexts (ISQD-SITA servers) to context queues (ISQD-SITA lanes) in accordance with constraints outlined in Section 4.4. When a physical context becomes idle (due to task completion or preemption),

the hardware scheduler selects the next virtual context from the head of a context queue by scanning for non-empty queues starting with the highest numbered lane (eldest tasks), based on the procedure explained in Algorithm 1. The scheduling hardware further tracks the execution time for each virtual context so that it can determine when the virtual context reaches the execution time limit imposed by the next ISQD-SITA scheduling cutoff. When this cutoff is reached, the task is preempted and the virtual context is descheduled and appended to the end of the context queue for the next ISQD-SITA lane. Tasks in the highest lane have no cutoff and will execute to completion.

CoreZilla provides a task-based software model where a single worker thread is pinned to each virtual context. The worker threads retrieve tasks from a single shared software task queue, as in an $M/G/k$ system, and manage both task queue synchronization and the CoreZilla scheduling hardware transparently to the executing tasks. Worker threads run the procedure shown in Algorithm 2. Each virtual context has an associated *elapsed time* that is maintained with the context and tracks how long the virtual context has been scheduled on a physical context since it began a new task. The elapsed time implicitly maps the context to an ISQD-SITA lane, based on where it falls relative to the ISQD-SITA cutoffs. An idle thread retrieves a task from the software task queue and resets the elapsed execution time for the virtual context to zero. The task then begins execution. When the elapsed execution time reaches the next ISQD-SITA cutoff, the context is preempted and appended to the context queue for the next ISQD-SITA lane. As noted in Section 4.4, we optimize for the special case where the next ISQD-SITA queue is empty and elide the context switch if the physical context would immediately reschedule the same virtual context. The central idea of our approach is to map a task-based software model to a thread-based execution model that allows the hardware to schedule among a fixed number of threads (virtual contexts) while managing a potentially unbounded number of tasks.

The scheduling hardware tracks the elapsed time for all physical contexts, and therefore also tracks the assignment of physical contexts to lanes, which can be inferred from the

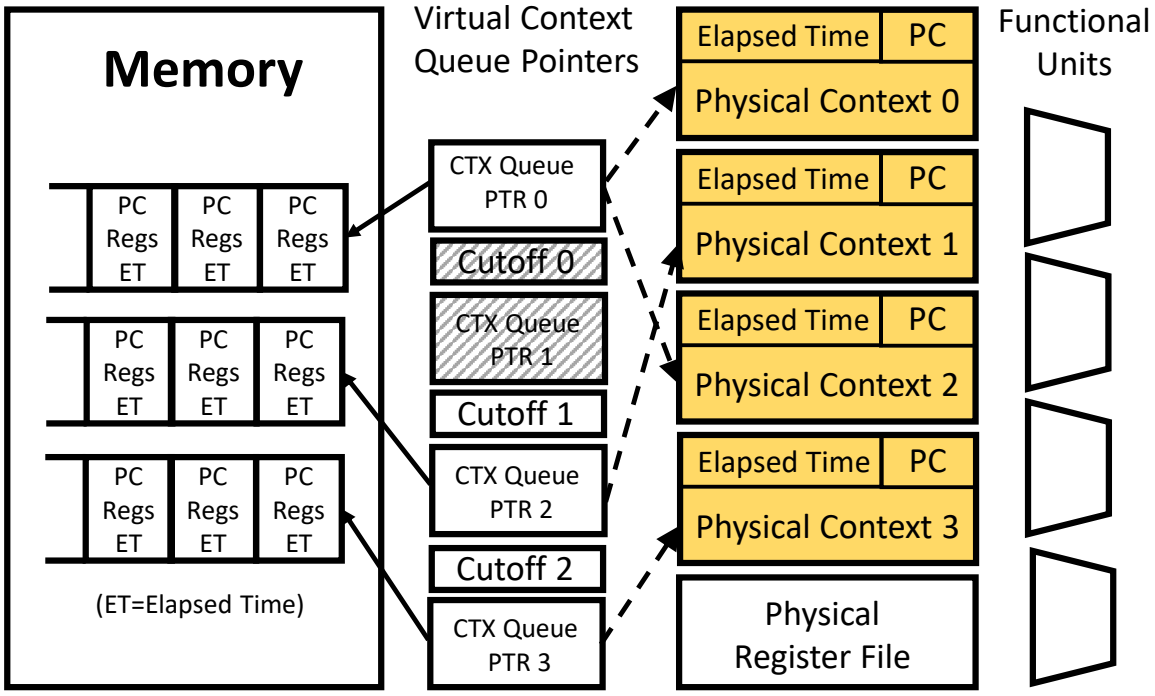


Figure 4.6: A 4-way CoreZilla with three context queues.

elapsed times and the cutoffs. Time can be maintained using any convenient monotonic counter (e.g., Intel's timestamp counter). Cutoffs are specified in a set of special registers and are set by the task framework based on prior knowledge or runtime monitoring of the service distribution. To enable ISQD-SITA, we add a mechanism that interrupts execution upon a write to a monitored memory location, to be able to detect new arrivals (highlighted line; only needed for ISQD-SITA). This scheme is similar to existing memory monitoring mechanisms, such as `mwait` in Intel processors [66], which detect changes to a memory location by tracking coherence invalidation messages.

Algorithm 2: High-level procedure of the worker threads

```
1 while true do
2     while task == nil do
3         reset_elapsed_time()
4         task = dequeue(task_queue)
5     end
6     async_monitor(task_queue)
7     run(task)
8     task = nil
9 end
```

Figure 4.6 illustrates the microarchitecture of a 4-way CoreZilla, with four physical

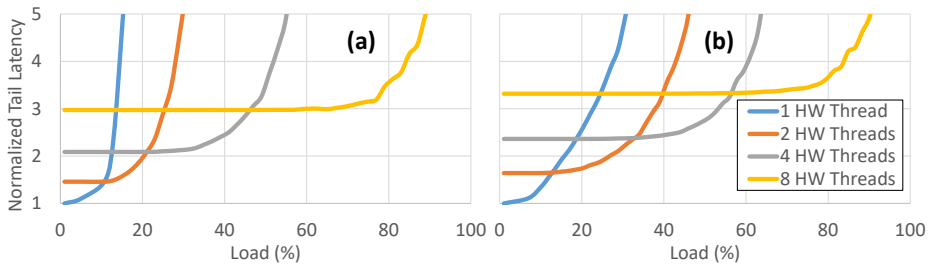


Figure 4.7: Normalized 99^{th} percentile latency at different loads for (a) Word Stemming and (b) McRouter.

contexts and three context queues (the number of reservations for lane 0 is two). Whereas there is fixed mapping between physical contexts and context queues in ESMT—leading to underutilization of physical contexts—in CoreZilla, the mapping changes dynamically, improving throughput and efficiency.

4.5.2 Automatic Load Adaptation

CoreZilla's two-level thread-context mechanisms enable hardware scheduling to implement ISQD-SITA. We add an additional mechanism to tune the number of active physical contexts to best serve the current load and service characteristics.

Some prior works [122, 28] advocate reducing or disabling hardware multithreading to avoid interference, which can exacerbate tail latency. Others [55, 94, 6] advocate hardware multithreading to improve core utilization and reduce Total Cost of Ownership (TCO). The guidance from these studies is in conflict.

We observe that the right number of hardware threads to balance tail latency and utilization depends on the workload's service time distribution and system load. When load is low, and end-to-end response time is dominated by service (rather than queuing) time, it is better to disable additional hardware threads, allowing a single thread to enjoy higher execution bandwidth and run faster without interference. However, as load increases, additional threads enable higher instruction throughput, which results in higher overall service rate at the core and reduced queuing time. Furthermore, as the service time distribution grows more heavy-tailed, additional threads become critical to minimize HoL blocking and

excessive queuing delays—high disparity service distributions are common in microservices [91, 162, 203, 185].

We illustrate these effects with an example. Figure 4.7(a) and (b) present end-to-end tail latency at different loads for Word Stemming and McRouter microservices (described in detail later in Section 4.7). As these figures show, fewer threads yield lower tail latency at low load, as each thread executes faster and queuing is rare at low load. However, as load increases, the better instruction throughput enabled by additional threads results in substantially lower queuing delay and tail latency. Furthermore, we observe that the break-even points for McRouter (b), which has a low-disparity service distribution, occur at higher relative loads compared to the break-even points of word stemming (a), which is a heavy-tailed microservice; word stemming requires more threads at lower load than McRouter to prevent HoL blocking.

To exploit this trade-off, CoreZilla incorporates an automatic load adaptation mechanism that dynamically tunes the number of physical contexts to minimize tail latency. CoreZilla's load adaptation comprises offline profiling and online adaptation phases. The offline profiling phase constructs a profile of tail latency vs. load across thread counts for a particular workload like those shown in Figure 4.7. The critical break-even points (crossings) in the load curves are then recorded in a lookup table.

During execution, the instantaneous arrival rate (the rate tasks are added to SQD-SITA lane 0) is monitored over 5-millisecond-scale windows to estimate load. Then, the lookup table is consulted to determine how many physical contexts to activate. A 5-ms window is long enough relative to the μs-scale service times of microservices to yield accurate load measurement, but also short enough to capture transient load changes as load fluctuations usually occur at least at the granularity of 10s of milliseconds [70, 187].

4.6 Discussion

Finding cutoffs. Finding SQD-SITA (and SITA) cutoffs is a non-trivial problem and is usually performed by empirical search. When the number of lanes is only two, cutoffs can be found by quantizing the service time distribution and linearly searching the entire space. However, with a larger number of servers/lanes, the search space grows combinatorially. Furthermore, our algorithm for finding cutoffs must consider server-ganged variants of SITA and SQD-SITA algorithms, where the lane count may be smaller than the server count. Hence, even the lane and reservation counts may not be fixed parameters when searching for the optimal cutoffs.

We propose a hill climbing-based heuristic for finding cutoffs. We empirically find the search space to be convex and the cutoffs to be near high quantiles of the service time distribution. We initialize the search with all reservations in the final lane (i.e., no cutoffs). We then consider moving one reservation to a new lane, searching for a tail-latency-minimizing cutoff value over descending quantiles of the service time distribution. We then iterate, considering (1) moving an additional reservation to the most recent lane, or (2) adding a new lane with a lower cutoff. The algorithm halts when neither moving a reservation nor adding a lane improves tail latency. Search time scales with the granularity of the search over quantiles and is independent of the number of servers/lanes. Hence, it is scalable to many servers. We performed point validations against exhaustive searches for a 4-server system and were not able to improve over this heuristic.

Comparison with SRPT and PS. Shortest Remaining Processing Time (SRPT) and Processor Sharing (PS) are other scheduling algorithms tailored for high-disparity service distributions. SRPT preempts a running task upon a new arrival and gives the execution resources to the new task if it is shorter than the remaining portion of the currently running task. While SRPT is proven to be asymptotically optimal [7, 152] (at most a constant factor worse than the optimal) for heavy-tailed service distributions, it can cause long tasks to starve and yield unpredictable results for tail latency, especially at high loads. Furthermore,

it is a size-based algorithm and is only applicable when task sizes are known or can be predicted accurately in advance [74] (much like SITA).

PS fairly shares execution capacity across all tasks by time sharing servers at small scheduling quanta. While PS is not size-based and does not starve long tasks, as in SRPT, it entails frequent preemptions (proportional to the task sizes), which hurt performance and may be impractical. We will show in Section 4.8 that (I)SQD-SITA outperforms both SRPT and PS, given enough servers, by isolating short tasks from long ones while respecting FIFO ordering of task arrivals.

Finite virtual contexts. Because CoreZilla supports only a finite number of virtual contexts, it is possible for all of them to be assigned tasks longer than the first (I)SQD-SITA cutoff (i.e., if 32 incomplete tasks all execute past the first cutoff). This scenario leads to a violation of both upper and lower-bound criteria, as there is no context able to execute newly arriving tasks in lane 0. Once any task completes execution, the hardware scheduler resumes obeying the (I)SQD-SITA constraints. The number of virtual contexts should be provisioned such that the probability that all virtual contexts are occupied by tasks longer than the first cutoff is negligible. This probability vanishes rapidly as the number of virtual contexts grows; 32 virtual contexts is more than sufficient.

Hardware costs and scalability. CoreZilla builds upon the ESMT hardware substrate, and from a hardware point of view, only extends it to have more than two physical contexts. The only hardware extension an N-way ESMT/CoreZilla requires over an N-way SMT core are N registers to hold virtual context queue pointers, N-1 registers to hold service cutoff points, N-1 timers/counters to measure elapsed times, and three registers to hold load break-even points. All of these structures in total add negligible area/power overheads to the datapath as they are small compared to the core's physical register file.Furthermore, all of these structures scale linearly with respect to the number of SMT execution lanes. Therefore, the main scalability bottleneck to add lanes is the SMT mechanism itself, because having more execution lanes complicates both the core frontend and backend, and particularly,

requires a larger register file to at least contain all the architectural registers of all physical contexts. Consequently, no commercial system supports more than 8 SMT threads; we have also only considered 2, 4, and 8 threads, which are the options available in IBM Power 8/9 microarchitectures that currently support the largest number of SMT threads. We have modeled these additional structures in McPAT [117] and found the area and power overheads of CoreZilla to be within 2% and 3% of a baseline SMT core, respectively. There is no accurate way to measure the clock frequency impact, except via RTL-level implementation and synthesis. However, we do not expect the additional control logic to be on the critical path and both the original Firmware Context Switching (FCS) [197] and subsequent designs employing this mechanism [138] report negligible cycle-time impact.

4.7 Evaluation Methodology

To evaluate SQD-SITA, we employ Stochastic Queuing Simulation (SQS), based on the BigHouse methodology and simulator [134]. We simulate until we achieve 95% confidence intervals of 5% error in reported results. We find cutoffs based on the heuristic explained in Section 4.6. We measure service time distributions of five microservices and feed their latency distribution histograms to our SQS framework. To accurately model the cost of preemption and context switches in CoreZilla and its alternatives, we model their hardware in the gem5 simulator [14] and include the preemption/restart latencies in our SQS experiments, following prior work [138]. We consider 2,4, and 8 SMT lanes (physical contexts) to model the available options for the number of hardware threads in IBM Power8/9 processors.

We use the following microservices for our evaluation:

- **FLANN:** we use a microservice benchmark based on

  FLANN [143], an open-source library for performing fast approximate nearest neighbor searches in high-dimensional spaces. FLANN uses Locality Sensitive Hashing (LSH) to perform k-nearest neighbor identification—a critical microservice employed

Table 4.1: Microarchitecture details of ESMT

Core	4-wide issue OoO, 192-entry ROB/PRF, 48-entry LQ, 32-entry SQ
SMT	ICOUNT [169] Fetch, up to 8 physical contexts, 32 virtual contexts
L1 cache	Private 64KB I/D, 64B lines, 2-way SA
LLC	1 MB per core, 64B lines, 8-way SA
Memory	50 ns access latency

in content-based similarity search. We use Google's Open Images dataset [64]. We consider variants of FLANN with (1) 20-bit, and (2) 12-bit LSH key sizes.

- **RocksDB:** We use RocksDB [53], a popular and widely deployed in-memory key-value store developed by Facebook. We use an open-source Twitter dataset [55] and RocksDB's default load generator with two different configurations, (1) where 90% of requests are GETS and 10% are SCANs, and (2) where 99% of requests are GETS and 1% are SCANs. SCAN requests scan 5000 keys and take approximately 50× longer than GETs.

- **Word Stemming (WS):** Stemming is a normalization process that reduces words to their root, employed in various cloud services, such as web search. We employ a word stemming microservice based on Oleander's implementation [153] of the Porter stemming algorithm [160, 161]. It is a high-disparity microservice as it hard-codes all stemming paths (prefixes, suffixes, etc.) into the control-flow and the length of paths for different words might be substantially different. Our queries include words from Wikipedia Redux [208].

- **Remote Storage Caching (RSC):** We implement a remote storage caching microservice as a simplified variant of existing host-side Flash caches [21, 105, 3, 80]. Our RSC microservice maps linear block addresses of a remote storage system to a local low-latency SSD using Cuckoo hashing [155]. We only consider read transactions. The three outcomes of a lookup query are that it might be a hit in the local memory,

hit in local SSD, or a miss.

- **McRouter:** We employ a consistent hashing microservice, based on Facebook's McRouter [118, 150], to route Key-Value (KV) operations to 100 leaf servers via a consistent hash function. We generate key-value lookup queries from an open-source Twitter dataset [55].

4.8 Results

4.8.1 SQD-SITA performance analysis

We first study alternative queuing organizations to gain insight into how each organization behaves in principle. In this section, we neglect the costs of preemption, and also consider scheduling algorithms that require task lengths to be known a priori, which is impractical for systems like CoreZilla. Figure 4.8 reports normalized 99^{th} percentile tail latencies achieved under various queuing organizations, including $M/G/k$, SITA, Ganged SITA (G-SITA), Preemptive Ganged SITA (PG-SITA), SQD-SITA, ISQD-SITA, Processor Sharing (PS), and Shortest Remaining Processing Time (SRPT). For PS, we consider a 2μs scheduling quantum. We consider 2, 4, and 8 servers, which correspond to (a), (b), and (c), respectively. For each number of servers in each workload, we set the offered load to the break-even point where our load adaptation system selects that configuration (e.g., break-even points in Figure 4.7).

As Figure 4.8 shows, ISQD-SITA improves tail latency by 2.28×, 3.39×, 4.76× over an M/G/k system with, 2, 4, and 8 servers, on average, respectively. Furthermore, while (I)SQD-SITA consistently improves tail latency over PG-SITA, there are a few cases where it falls short of the tail latency achieved by (G-)SITA. These cases arise because non-preemptive SITA is a size-based algorithm, which can exploit its prior knowledge of task sizes. In addition, note the impact of server ganging—with 4 and 8 servers, SITA yields significantly higher tail latencies than G-SITA, which, in many cases, are even higher than those of the

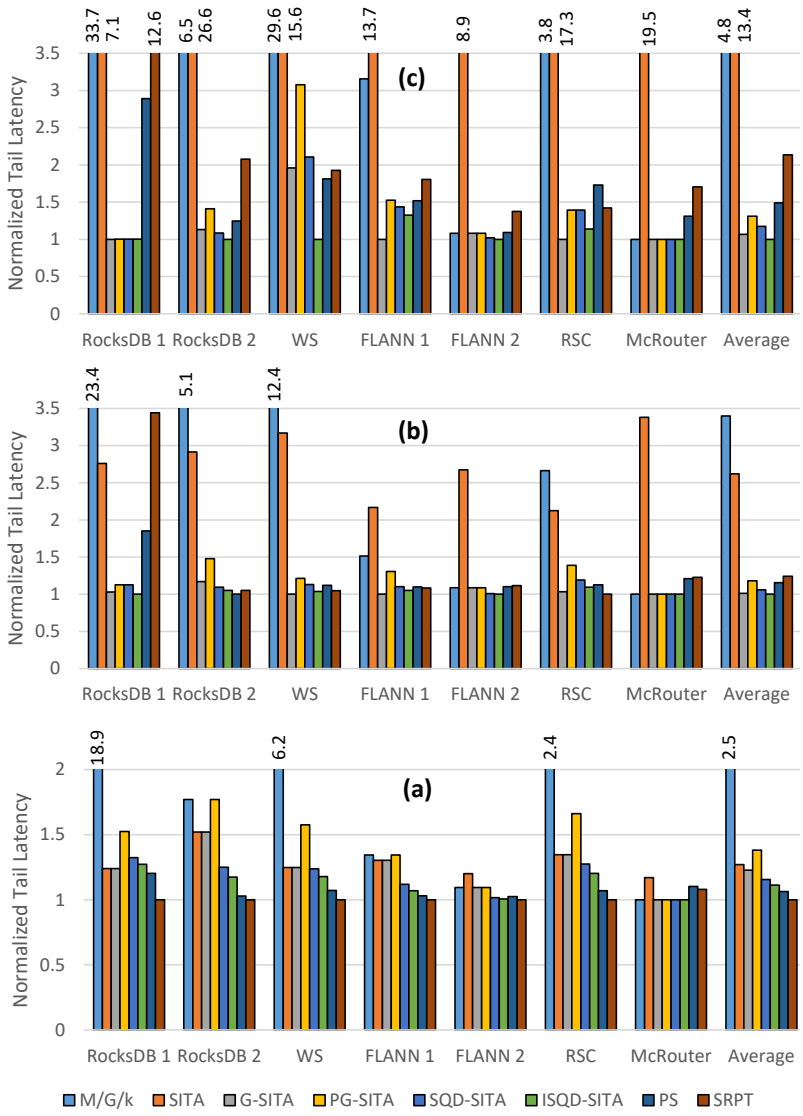


Figure 4.8: Normalized 99^{th} percentile latency under various organizations for (a) 2, (b) 4, and (c) 8 servers.

baseline M/G/k system. Having too many lanes results in unnecessary load imbalance in systems like SITA, which, unlike SQD-SITA, statically assign servers to lanes; only a few lanes are sufficient to eliminate HoL blocking, regardless of the number of servers.

We observe that, with 8 servers, ISQD-SITA consistently outperforms PS and SRPT (by 49% and 2.13 $\times$, on average), and with 4 servers ISQD-SITA consistency outperforms PS and is outperformed by SRPT only for RSC workload (16% and 24% average improvement over PS and SRPT). However, with two servers, ISQD-SITA is outperformed by PS and

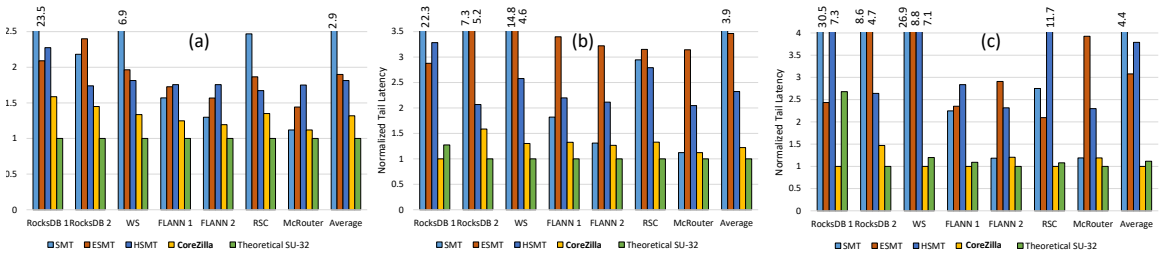


Figure 4.9: Normalized 99^{th} percentile latency of CoreZilla and alternatives for (a) 2, (b) 4, and (c) 8 hardware threads.

SRPT by 6% and 11%, on average, respectively. Neither PS nor SRPT respect FIFO ordering of the tasks; PS fairly shares the system capacity among all tasks and SRPT strictly prioritizes short tasks. These algorithms are well-suited only for high-disparity service distributions and fall short of a FIFO-ordered (M/G/k) system for low-disparity distributions [19, 207] (e.g., McRouter). However, with enough servers, ISQD-SITA isolates short tasks from long ones while respecting their FIFO arrival ordering, outperforming PS and SRPT.

4.8.2 CoreZilla performance analysis

In this section, we seek to find the optimal core microarchitecture, and therefore only consider non–size-based scheduling policies, which can be practically adopted by a system like CoreZilla, and also consider the costs of preemption. Figures 4.9(a), (b), and (c) report normalized 99^{th} percentile tail latencies achieved by different core microarchitectures with 2, 4, and 8 physical SMT contexts, implementing different scheduling policies. We compare SMT, Hierarchical SMT (HSMT) [138], Express-Lane SMT (ESMT) [140], and CoreZilla, with the same number of physical contexts (2/4/8). HSMT effectively implements PS scheduling by time multiplexing all virtual contexts on the physical contexts at 2μs time quanta. We also compare against a theoretical 32 cores scale-up organization—in all other designs, there is a distinct task queue per physical core.

CoreZilla improves tail latency over a conventional SMT core and an ESMT core with 2, 4, 8 physical contexts by 2.25×, 3.23×, 4.38×, and 38%, 2.83×, 3.07× on average, respectively. Improvements are slightly smaller than reported in the previous section as

the costs of preemption are now included. Also it is notable that whereas ESMT achieves only slightly higher tail latency than CoreZilla with 2 physical contexts, it falls well short of CoreZilla's performance with more physical contexts. ESMT suffers significantly from underutilization of physical contexts as it statically maps each physical context to a context queue. Server ganging and dynamic reallocation of physical contexts to context queues in (I)SQD-SITA solve this problem in CoreZilla. Moreover, note that while ESMT and HSMT may result in tail latencies that are higher than SMT, CoreZilla never falls short of SMT performance, thanks to the same mechanisms.

CoreZilla significantly outperforms HSMT (which implements PS), since the number of preemptions incurred under PS are much higher than under (I)ISQD-SITA; with PS, each task requires, on average, $mean-task-size/scheduling-quantum$ preemptions ($mean-task-size$ can be 10s – 100s of microseconds). However, with SQD-SITA, each task incurs at most one preemption per cutoff/lane. Unlike SQD-SITA, the number of preemptions under ISQD-SITA is not bounded. However, as these results show, the net gain is always positive.

Finally, we observe that, with two and four physical contexts, CoreZilla achieves 99^{th} percentile tail latency that is within 31% and 22% of a theoretical 32-core scale-up organization, respectively. With eight physical contexts, however, due to superior task scheduling, CoreZilla is even able to achieve 12% lower average tail latency compared to a theoretical 32-core scale-up organization, obviating the need for having a cross-core shared queue and its attendant overheads.

4.8.3 Impact of preemptions in ISQD-SITA

As noted before, ISQD-SITA may incur additional preemptions compared to SQD-SITA. Figure 4.10 reports the average number of preemptions in 8-server Preemptive-SITA, SQD-SITA, and ISQD-SITA systems. Here we consider non-ganged (i.e., 8-lane) variants of these algorithms, as these incur the most preemptions. Whereas ISQD-SITA increases the

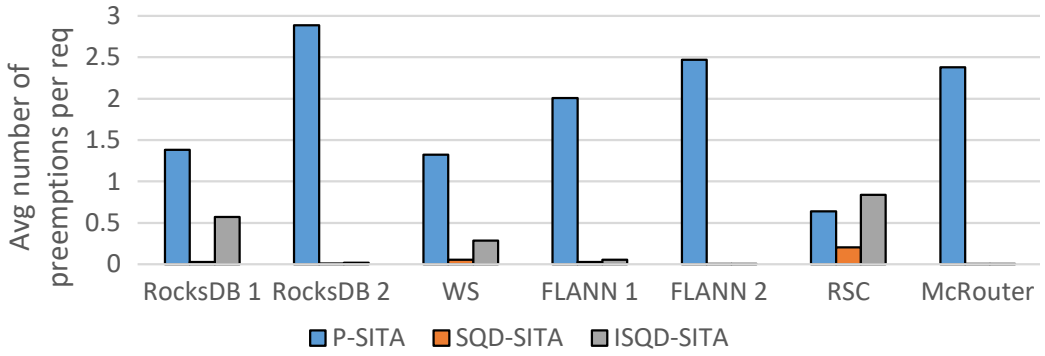


Figure 4.10: Average number of preemptions per request for different scheduling policies in various microservices.

number of preemptions relative to SQD-SITA, it often significantly reduces their number compared to Preemptive-SITA. Interestingly, the average number of preemptions per task of ISQD-SITA exceeds that of Preemptive-SITA for the RSC workload (0.84 vs. 0.63) because RSC includes both exceptionally long and exceptionally short tasks. In this scenario, before any nominal or long task finishes, one task that violates the upper bound criterion may be be preempted and resumed multiple times, due to rapid arrivals of such short tasks. Nevertheless, as shown in Figure 4.10, the overall average number of preemptions in (I)SQD-SITA is negligible.

4.9 Related Work

The most related work to ours is RPCValet [36], which proposes a potential solution for approximating a scale-up queuing organization on a scale-out system in the presence of on-chip integrated NICs. With RPCValet, instead of the integrated NIC "pushing" packets into each core's dedicated queues, which may result in load imbalance and HoL blocking, each core "pulls" a packet from the NIC once it is done processing the previous packet. The single shared packet queue is managed in hardware by the on-chip NIC and distributes packets into the cores' local queues. RPCValet's solution is only applicable to systems with on-chip integrated NICs and can at best achieve the lower-bound tail latency of a theoretical scale-up system. However, as we showed in Section 4.8, with sufficient physical contexts,

CoreZilla can reduce the tail latency even beyond that of a theoretical 32-core scale-up system because CoreZilla augments the queuing model with the (I)SQD-SITA scheduling policy, which is inherently able to isolate long and short tasks and prevent HoL blocking.

A large body of prior work seeks to lower the tail latency of interactive services. However, most past studies target classic monolithic services with millisecond- to second-scale service times and, hence, require different approaches for our target microservices. We discuss various classes of such studies:

Parallelization and Heterogeneity. One class seeks to accelerate long tasks by parallelizing them on multiple cores to reduce their processing time and queuing impact. Jeon et al. [88] propose an adaptive solution to determine the required degree of parallelism for each query based on the offered load. In a follow up work [89], they propose a feature-based prediction model to predict long tasks and parallelize them. Haque et al. [70] propose an incremental approach that increases the degree of parallelism as a task advances in execution. In a follow up work [71], they move the longer tasks to faster cores in heterogeneous platforms to accelerate them further. All of these techniques are only applicable to ms-scale services that are easily parallelized, such as web search.

Voltage/Frequency Boosting. Another class seeks to boost core voltage and frequency to accelerate long tasks. Adrenaline [82] considers SET requests in a key-value store as long tasks and accelerates them; their approach is application-specific and not easily generalizable. Rubik [97] takes a more general approach and by probabilistically accelerating queries based on the service time distribution and the position of each query in the queue. However, it fails to capture heavy-tailed service distributions, where the probability of HoL blocking is high, and relative position of queries in the queue has low correlation with their queuing time.

Non-FIFO Scheduling. Another class seeks to minimize tail latency of high-disparity services by employing better-than-FIFO scheduling schemes to eliminate HoL blocking. Various authors [116, 162] propose per core task queues augmented by work-stealing to improve tail latency. While work-stealing is an effective approach for improving utilization

and throughput, it is not well-suited for server applications, where the objective is to minimize the response-time. Work-stealing allows cores to take tasks from other queues when their own queue is empty to improve utilization; it does not solve the HoL blocking problem within each queue.

Shinjuku [91] proposes to address tail latency by employing PS for high-disparity task distributions. As we have shown, (I)SQD-SITA usually outperforms PS as PS does not respect the FIFO ordering of task arrivals. Baraat [43] proposes a FIFO with limited parallelism (FIFO-LM) scheme, wherein a number of oldest tasks (e.g., 8) are time-multiplexed but younger tasks wait for them in a FIFO queue. Interestingly, this mechanism is already implemented in SMT cores as the active threads (which serve the oldest tasks) are truly sharing the processor. CoreZilla outperforms conventional SMT designs which implement such a policy.

Size-based Scheduling. Finally, a few prior studies propose size-based scheduling mechanisms by correlating the processing time of a request with one of its features. Harchol-Balter et al. [74] propose to use SRPT for file and web servers by estimating request processing times based on file sizes. Didona et al. [42] propose a cross-core sharding of key-value store queries, based on object sizes. Their approach effectively implements server-ganged SITA across cores by estimating task lengths based on object sizes. These approaches are only applicable to their respective services and are not comparable to generic approaches like SQD-SITA.

4.10 Conclusion

In this chapter, we proposed *Q-Zilla* as a scheduling framework and its hardware instantiation to tackle the tail latency of microservices. Q-Zilla is composed of *Server-Queue Decoupled – Size-Interval Task Assignment* (SQD-SITA), as a tail-aware scheduling algorithm, and Interruptible SQD-SITA (ISQD-SITA) which further improves tail latency at the cost of additional preemptions. (I)SQD-SITA dynamically reallocates servers to lanes to increase server utilization with no performance penalty. Finally, we proposed *CoreZilla*, as a hardware realization of (I)SQD-SITA in a multithreaded core. CoreZilla improves tail latency over a conventional SMT core with 8 threads by 4.38$\times$ and outperforms a theoretical 32-core scale-up organization by 12%, on average.

CHAPTER V

Parslo: A Gradient Descent-based Approach for Partial SLO Allocation in Virtualized Cloud Microservices

5.1 Introduction

There is a growing trend towards building modern cloud services using microservice architectures, wherein a complex application is decomposed into tens to hundreds of independent, loosely-coupled, distributed components—in form of a Directed Acyclic Graph (DAG) [59, 183]. Microservice architectures have been adopted by major cloud-based companies, such as Facebook, Netflix, Linkedin, etc., as they significantly improve programmability, reliability, manageability, and scalability of cloud services. For example, a Facebook news feed service query may flow through a chain of microservices such as Sigma (a spam filter), McRouter (a protocol router), Tao (a distributed social graph data store) and MyRocks (a user database) [183]. Figure 5.1 illustrates two open-source microservice DAGs from [59], representing a social network and a media service.

Service Level Objectives (SLOs) impose bounds on the average or tail of the end-to-end latency distribution in a cloud service, to ensure an acceptable level of service quality and user satisfaction [139]. Auto-scaling frameworks, such as Google's Autopilot [171], continuously monitor the response time of the incoming requests to a service and upsize or downsize the service by increasing or decreasing the number of instances (VMs or

containers) in the virtual cluster to meet the latency SLO at minimal cost [164]. However, with microservice-based applications, it is unclear which microservices need to be scaled when end-to-end latency SLOs are violated.

Most existing systems impose partial latency SLOs on individual microservices so as to ensure that the end-to-end SLO is met if all partial SLOs are met [198, 129]. However, these partial SLOs are usually allocated empirically, which may significantly increase the total deployment cost of the service [164, 95]. Recent research studies propose to use centralized Machine Learning (ML)-driven auto-scaling frameworks to determine which microservice(s) to scale when end-to-end latency SLOs are violated [60, 163]. While these frameworks can react dynamically to changes in the microservice DAG topology, they are heavy-weight and need frequent data collection and retraining. Furthermore, whereas such frameworks outperform empirical partial SLOs, they do not guarantee optimality and may still result in suboptimal service deployment costs.

In this chapter we propose Parslo—a Gradient Descent-based approach to allocate partial SLOs to different nodes of a microservice graph under an end-to-end latency SLO. Parslo isolates different microservice nodes within a DAG from one another and enables each microservice to be scaled independently through its own auto-scaling framework. At a high level, the Parslo algorithm breaks the end-to-end SLO budget into small "SLO units", and iteratively allocates one SLO unit to the best candidate microservice to achieve the highest total cost savings until the entire end-to-end SLO budget is exhausted. Parslo employs novel mechanisms to be applicable to general microservice DAGs—as such the ones depicted in Figure 5.1—which may include microservice dependencies, branching path, as well as parallel indexing and sharding.

To the best of our knowledge, Parslo is the first systematic partial SLO allocation scheme for microservices supporting general microservice DAGs. Parslo achieves the optimal partial SLO allocation, thereby minimizing the total deployment cost for the entire service. Our evaluation results demonstrate that Parslo reduces service deployment costs by more than

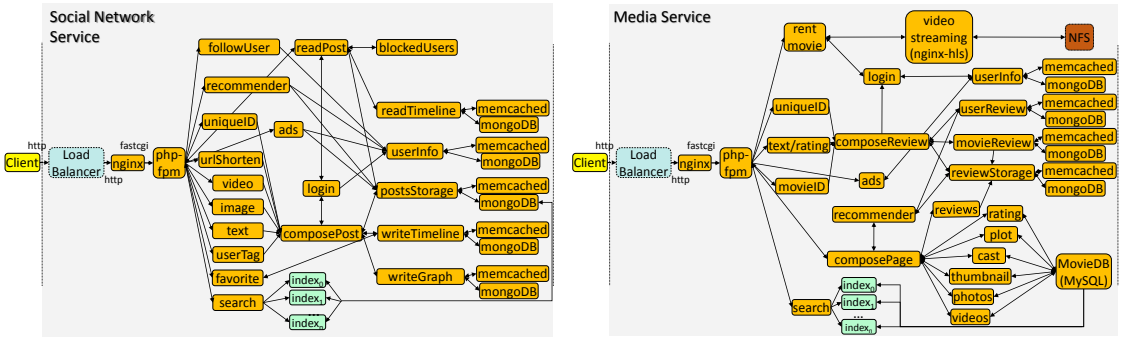


Figure 5.1: The microservice DAG for a social network and a media service system from [59].

.

6× in microservice-based applications, compared to a state-of-the-art partial SLO allocation scheme.

5.2 Background and Motivation

5.2.1 SLOs and Auto-Scalers

Service Level Objectives (SLOs) specify key properties of cloud applications, such as availability, latency, etc. One of the key features of cloud-based services is their *elasticity*, wherein the number of instances allocated to the service can be quickly tuned up or down (i.e., the services is upsized or or downsized) depending on the current request arrival rate, to ensure the end-to-end latency SLO is met. Latency SLOs are usually defined based on the average or (e.g., 99^{th}-percentile) tail latency. Tail latency is the preferred metric in defining latency SLOs as it bounds the slowest interactions of users with the service and guards overall user satisfaction.

Figure 5.2 illustrates the high-level operations of a reactive auto-scaler, wherein the end-to-end response time of a service is continuously monitored. If the observed response time exceeds the latency SLO, the service is iteratively upsized (i.e., by adding service instances). Spreading the load over more instances reduces the queueing delay at each instance until the latency SLO is met. Conversely, if the end-to-end latency is significantly

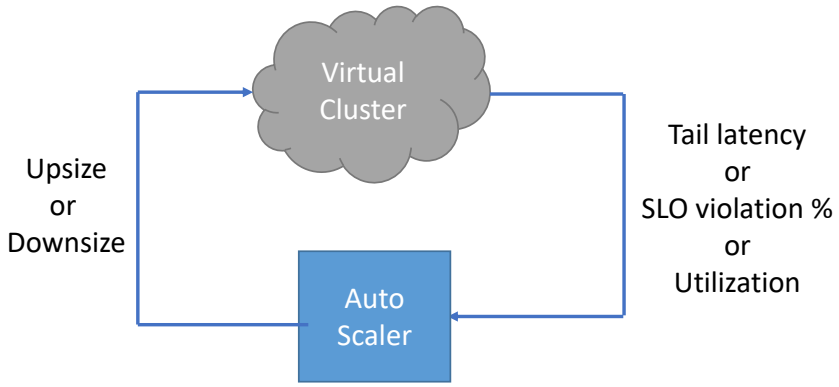


Figure 5.2: High-level operations of an auto-scaling framework.

lower than the SLO, the service is downsized to close the latency gap and meet the latency SLO while reducing cost (i.e., fewest active instances).

When SLO is defined based on the tail (rather than average) latency, it is preferred that each instance reports to the auto-scaler the fraction of requests that violate the SLO's latency target, rather than the raw tail latency, as this metric is easier to aggregate across instances. However, a more efficient approach is to profile the service offline in one instance across the load spectrum and derive the load-latency profile of the service—which we call a "hockey stick" graph for its distinctive shape—shown in Figure 5.3(a). With this approach, the maximum utilization level at which each instance may operate without violating the latency SLO is identified. Therefore, if an instance's utilization exceeds this threshold, the auto-scaler may proportionally increase the number of instances, rather than incrementally upsizing the service until the latency SLO is met. This approach is particularly advantageous in response to load spikes or flash crowds where incremental scaling may take a long time to converge and large queues might build. Furthermore, this approach is easier to implement in automated cluster management frameworks—such as Kubernetes—as these frameworks typically scale the services based on the CPU utilization of the instances since this metric externally available to the cluster manager, as opposed to raw latency or SLO violation rate, which need to be communicated to cluster manager by the code running within the instance.

Flash crowds or load spikes refer to brief periods of sudden, significant increases in the

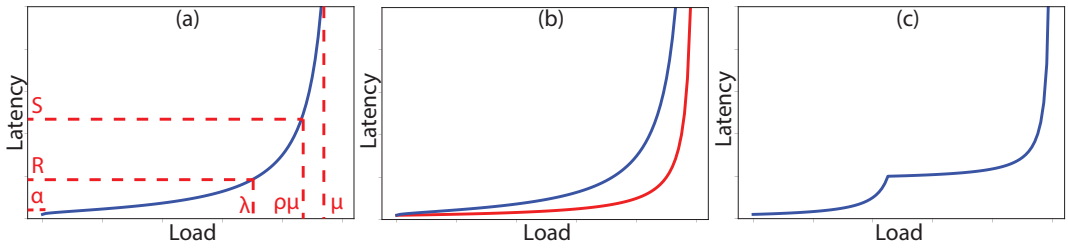


Figure 5.3: (a) load-latency profile of a microservice, known as the "hockey-stick" graph (λ: an arbitrary arrival rate; R: the response time corresponding to λ; α: zero-load latency; μ: maximum load; S: latency SLO; ρ: maximum utilization without violating the SLO. (b) two different valid hockey-stick graphs, based on an $M/M/1$ (red) and and $M/G/1$ queuing model with a heavy-tailed service-time distribution (blue). (c) an invalid hockey-stick graph.

request arrival rate. During load spikes, the arrival rate may go up by 20% or even 2× [10], for a short period of time. Load spikes present a significant challenge to auto-scalers, as increasing the number of instances takes time, and delayed reaction to load-spikes results in large queue build-ups within the instances, whose effect may persist for a long time after the load spike subsides. To address this issue, auto-scalers typically over-provision the number of instances, in anticipation of load spikes. Whereas this may result in wasted resources, it prevents load spikes from building long queues within the instances, which result in drastic increases in SLO violation rate. Sophisticated proactive auto-scalers seek to mitigate this issue by over-provisioning the right number of additional instances via estimating the spike magnitudes based on previous spike behavior [61].

5.2.2 Latency SLOs for Mircoservices

Modern services are implemented as DAGs of loosely-coupled microservices. A key challenge with microservices is that is the latency SLOs are usually defined based on the the whole-service end-to-end response time, rather than individual microservices, given the goal of the latency SLOs is to optimize the end user experience. Hence, with microservices, it is unclear how each individual microservice must be scaled, when the end-to-end latency SLO is violated.

There are two general approaches to address this issue: (1) most existing and deployed systems as well as classic research proposals assign *Partial SLOs* to individual microservices [198, 129], to ensure each microservice can be independently managed and scaled through its own auto-scaling framework. This approach is cheap and straightforward especially because different microservices may be implemented atop different cluster management frameworks, which are not easy to integrate. However, existing systems assign partial SLOs to microservices through ad hoc and empirical mechanisms, which may result in a sub-optimal total cost (i.e., total number of instances). (2) Some recent research proposals advocate end-to-end auto-scaling frameworks, wherein a single controller scales the number of instances for all microservices upon traffic changes, using machine learning-based techniques [60, 163]. Whereas these schemes are able to quickly react to changes in the microservice graph topology, they impose a significant overhead for implementing an ML-driven centralized controller, they are not trivially scalable to large microservice graphs, they require frequent data collection and retraining, and most importantly, there is no guarantee of optimality for such systems—i.e., they do not necessarily achieve a minimal cost to meet the end-to-end latency SLO.

In this chapter, we take the simpler approach of allocating partial SLOs to individual microservices but seek to propose a systematic mechanism for achieving an optimal solution, rather than empirically allocating partial SLOs. Our goal is to design an algorithm that takes the microservice DAG and the hockey-stick graph of each node as an input and produces optimal SLOs for each microservice, in a way that meeting these partial SLOs results in the fewest total number of instances while ensuring the end-to-end latency SLO is met.

5.2.3 Optimal Partial SLO Allocation

As noted earlier, prior works that propose partial latency SLOs often allocate them empirically, which may result in suboptimal costs. A recent work [95] considers chains of microservices, as shown in Figure 5.4(a), and proposes to divide the end-to-end latency SLO

across microservices proportional to their average service time, as stated in Equation 5.1. In this subsection, we seek to understand whether such an allocation optimizes the total cost for the end-to-end service.

$$SLO_k = \frac{T_k}{\sum_{i=1}^{n} T_i} \tag{5.1}$$

We consider the following parameters for each microservice, as shown in Figure 5.3(a): α specifies the zero-load latency of the microservice—it can can be the average or the tail of the service time, depending on the metric of interest for the latency SLO. μ specifies the maximum arrival rate that an instance of the microservice can sustain (without incurring infinite queuing). c specifies the cost of a single instance—this parameter may be set to 1 when all microservices are deployed on the same instance type. However, since typically each microservice may be deployed on a different instance type with different configurations (cores, memory, etc.), it is more realistic to optimize for the total cost rather than the total number of instances. Finally, the ϕ function specifies the shape of the hockey-stick graph for the microservice. In other words, ϕ takes as input a number within the $[0,1]$ range that specifies the fraction of the maximum load (μ) and produces the latency of the instance, normalized to the zero-load latency α.

Based on the defined parameters, a microservice's response time corresponding to a particular arrival rate λ can be inferred from Equation 5.2. Similarly, if we set the partial latency SLO of a microservice to be s, the maximum utilization of an instance is calculated by Equation 5.3. As a result, the relative cost of the microservice is given by Equation 5.4 (the number of instances is inversely proportional to the utilization of each instance).

$$(Response\ time)\ \ R = \alpha\phi(\frac{\lambda}{\mu}) \tag{5.2}$$

$$(Max\ utilization)\ \ \rho = \phi^{-1}(\frac{s}{\alpha}) \tag{5.3}$$

$$Microservice\ cost\ = \frac{c}{\mu\rho} = \frac{c}{\mu\phi^{-1}(\frac{s}{\alpha})} \tag{5.4}$$

To examine the optimality of the SLO allocation scheme presented in Equation 5.1, we consider a chain of only two microservices. Equation 5.5 denotes the total cost for the chain, assuming both microservices are operated at their individual maximum utilization. To derive the optimal partial SLO allocation, we set $s_2 = SLO - s_1$ (SLO denotes the end-to-end latency SLO) and find the roots of the derivative of the total cost over s_1, which results in Equation 5.6. As this equation clearly shows, allocating partial SLOs proportional to zero-load latencies of the microservices (i.e., $\frac{s_1}{\alpha_1} = \frac{s_2}{\alpha_2}$) can be a solution but only under particular assumptions: (1) the shape of the hockey-stick graph for the two microservices is the same (i.e., $\phi_1 = \phi_2$), (2) the cost of each instance for the two microservices is equal (i.e., $c_1 = c_2$), and (3) zero-load latencies of the microservices are inversely proportional to their maximum load (i.e., $\alpha_1\mu_1 = \alpha_2\mu_2$). Note that (3) is often true when latency SLOs are defined based on the average latency but not necessarily true when they are defind based on the tail latency. In such cases, allocating partial SLOs based on either the average or the tail of the service times (zero-load latency) does not minimize the cost.

$$Total\ cost\ = cost_1 + cost_2 = \frac{c_1}{\mu_1\phi_1^{-1}(\frac{s_1}{\alpha_1})} + \frac{c_2}{\mu_2\phi_2^{-1}(\frac{SLO-s_1}{\alpha_2})} \tag{5.5}$$

$$\frac{c_1}{\alpha_1\mu_1}\frac{\phi_1^{-1\prime}(\frac{s_1}{\alpha_1})}{\phi_1^{-1}(\frac{s_1}{\alpha_1})^2} - \frac{c_2}{\alpha_2\mu_2}\frac{\phi_2^{-1\prime}(\frac{SLO-s_1}{\alpha_2})}{\phi_2^{-1}(\frac{SLO-s_1}{\alpha_2})^2} = 0 \tag{5.6}$$

To further illustrate the lack of optimality for the SLO allocation approach described in Equation 5.1, we consider the hockey-stick graphs presented in Figure 5.3(b) for the two microservices in the chain. In this case, even if the all other parameters are equal for

the two microservices (i.e., α, μ, c) the optimal SLO allocation may result in up to 24% lower cost, compared to the allocation approach presented in Equation 5.1. Aside from lack of optimality, this approach is useful only for chains of microservices and does not generalize to arbitrary DAGs as seen in real applications. Our goal is to derive a partial SLO allocation scheme, which, given an end-to-end latency SLO budget, results in an optimal allocation—minimizing the total cost—and is applicable to the general microservice DAGs of real applications.

5.3 Parslo: SLO Allocation

In this section, we describe the *Parslo* partial SLO allocation methodology. Parslo is an iterative optimization algorithm based on the Gradient Descent theory. Parslo takes as input an end-to-end latency SLO, a microservice DAG, and the load-latency profile (i.e., the hockey-stick graph) for all of the nodes in the DAG, and produces an optimal partial SLO allocation that minimizes the deployment cost for the end-to-end service. Even though Gradient Descent is only guaranteed to converge to a local optimum, since the cost function Parslo seeks to optimize is convex, Parslo guarantees to find the globally optimal solution. Parslo is applicable to general microservice DAGs that may be seen in real applications, such as the ones shown in Figure 5.1. We first describe a simple variant of the algorithm for a chain of microservices, shown in Figure 5.4(a). Then, in the following subsections, we will describe how the algorithm handles more complex scenarios that may be found in microservice DAGs.

Algorithm 1 presents pseudocode for the Parslo partial SLO allocation methodology. At a high level, the algorithm breaks the end-to-end SLO budget into small "SLO units". At each step, the algorithm allocates one marginal SLO unit to the best candidate microservice to achieve the highest total cost savings. The algorithm iteratively repeats this process until the entire end-to-end SLO budget is exhausted. The size of the SLO unit must be small enough to ensure the algorithm does no jump over the optimal solution. We empirically find

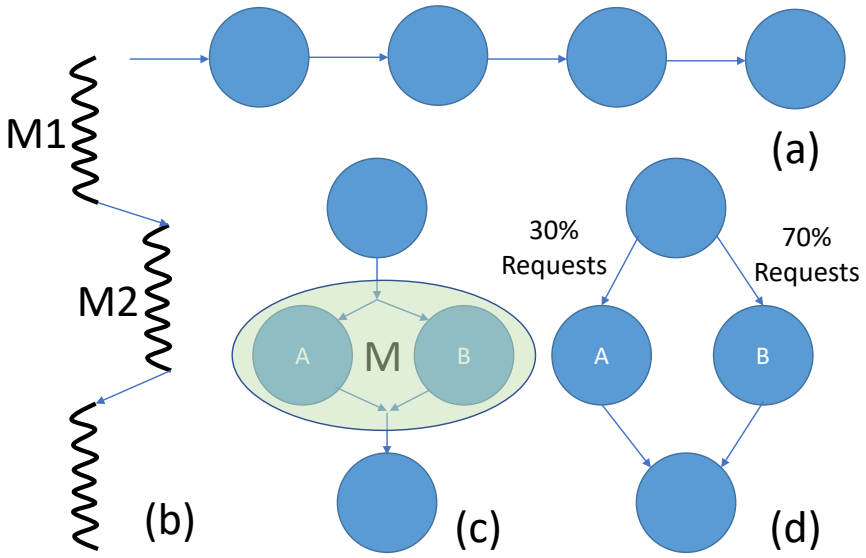


Figure 5.4: (a) a chain of microservices, (b) dependencies across microservices, (c) parallel indexing and sharding, (d) branching paths.

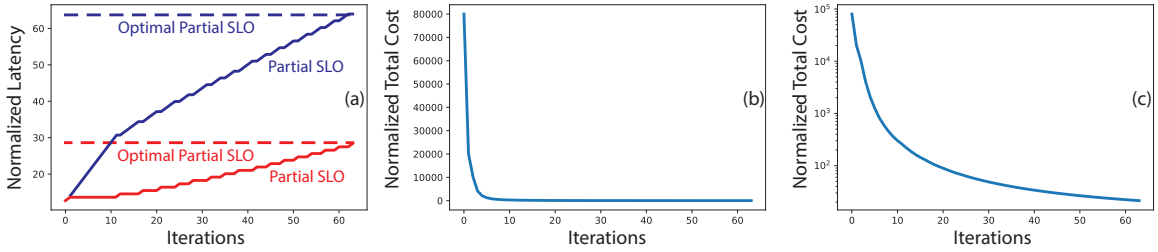


Figure 5.5: (a) marginal increase of Partial SLOs with Parslo. Final values match the optimal partial SLOs found by exhaustive search. (b) incremental reduction of total costs shown in linear, and (c) log scale.

that dividing the SLO budget into 1000 SLO units is sufficient for converging to an optimal solution.

The algorithm first initializes all microservices by allocating a single SLO unit to them. This step ensures the algorithm starts with a finite total cost, so the cost saving can be calculated at each step. As a result, the algorithm starts with an SLO budget that is smaller than the end-to-end SLO, since a fraction of the budget is allocated during initialization. Then, the algorithm finds the microservice that yields the *highest cost savings* if allocated one more unit of SLO. The cost of a microservice node given a particular

partial SLO is calculated using Equation 5.4. The algorithm continues allocating SLO units one at a time until the entire end-to-end SLO budget is exhausted. As a result, the computational complexity of the algorithm is linear in the number SLO units and the number of microservices in the chain. Finding the best microservice for each incremental SLO unit can be optimized to logarithmic complexity via priority queue-based implementations, but this optimization yields little gain as the number of microservices in real applications is usually less than 100. In our experiments, the algorithm converges to the optimal solution in less than a minute, even with complex microservice DAGs (which we will discuss later).

```
1 SLO-budget = Initialize()
2 while (SLO-budget > SLO-unit){
3    best-candidate = find-candidate-with-highest-cost-saving()
4    best-candidate.increase-SLO(SLO-unit)
5    update-total-cost()
6    SLO_budget=SLO-budget-SLO-unit
7 }
8
```

Algorithm V.1: Parslo SLO Allocation Process

Figure 5.5 illustrates the Parslo algorithm for a chain of two microservices. As shown in the figure, both microservices are initially allocated a single SLO unit. Then, their partial SLOs are incrementally increased, based on the marginal resource savings, until the entire SLO budge is allocated. Final partial SLOs match the optimal values found by exhaustive search. Figures 5.5 (b) and (c) report the total deployment cost as the partial SLOs change. As shown in these Figures, the total cost initially reduces rapidly but the reductions slow down as the sensitivity of the hockey-stick graphs to utilization is much higher when latency constraints are tight.

A key requirement for the algorithm to yield a globally optimal solution is that the slope of the hockey-stick graphs of all microsevices must always be increasing (i.e., the second derivative of the function is always positive). This requirement ensures that the

optimum resource saving decreases monotonically across iterations, resulting in a convex cost function. As a result, the cost function only has a single local optimum, which is the globally optimum solution. As an example, the algorithm doesn't necessarily yield an optimal SLO allocation if one of the microservices exhibits a hockey-stick graph similar to 5.3(c). However, this assumption is generally true in almost all microservices as the response time increases at a higher pace when the load increases in almost all queuing organizations [139]. We next describe how Parslo handles more complex scenarios that arise in real microservice DAGs.

5.3.1 Microservice Dependencies

We described the basic operation of Parslo for a chain of microservices. However, many microservices exhibit dependencies, wherein a non-leaf microservice issues subsequent requests to another microservice while processing a request and is only able to continue processing the request after it receives a response from the next-tier microservice, as shown in Figure 5.4(b). In such scenarios, the response time of a request at the first microservice (M1) depends upon the response time at the second microservice (M2).

To understand the impact of such dependencies on SLO allocation, we briefly explain different implementation alternatives for microservice M1 [187]: (1) In the most straightforward way, M1 may be implemented *synchronously*, wherein each core processing a request issues a subsequent request to M2 and waits until the response arrives back. During this period, the core remains idle even if there are outstanding requests in the queue. (2) To optimize for utilization and throughput, M1 may instead be implemented in an *over-subscribed* manner, wherein the number of processing threads is larger than the number of cores. In this case, when a thread is stalled, waiting for the next-tier response, it is context switched with another thread to process another request and keep the core busy. Finally, (3) M1 may be implemented *asynchronously*, wherein the number of threads is the same as the number of cores but each thread picks a new request after issuing the subsequent request

to M2 and continues processing the previous request when the response from M2 returns, using an explicit state machine to track the partial progress of the requests. Asynchronous implementations are significantly more complicated than over-subscribed implementations but are more efficient as they do not incur context switching overheads.

A key requirement in Parslo is that different microservices must exhibit hockey-stick graphs that are independent of one another. When a microservice is implemented in an over-subscribed or asynchronous manner (i.e., options (2) or (3) above) its throughput behavior would be independent of the next-tier microservice latency as cores do not remain idle while a request is waiting for a response. To isolate the latency behavior from the next tier, we define the response time of a request as the time a request has spent waiting or processing within a microservice, excluding the time a microservice waits for the next-tier response. In other words, in Figure 5.4(b), the response time of microservice M1 at a given load (represented by its hockey-stick graph) corresponds to the time a request is enqueued waiting for service at M1 plus the time it is being processed, but exclusive of the time waiting for a response from M2—the time spent at M2 is subtracted from the total response time of M1 when measuring the hockey-stick graph. Parslo performs its SLO allocation based on this modified definition of response time for dependent microservices.

As a result, using our modified definition of the response time, microservice dependencies do not impact partial SLO allocation for over-subscribed or asynchronous deployments, since both the throughput and latency behaviors of the calling microservice are independent of the latency at the next-tier microservice. Hence, Parslo is applicable only for such deployment approaches. Fortunately, most real applications use one of these deployment approaches as a naive synchronous implementation is highly inefficient [187].

5.3.2 Parallel Indexing and Sharding

Another common pattern in real microservice-based applications is a fan-out communication pattern with concurrent batched retrieval from numerous shards; this pattern arises in

search use cases, as shown in Figure 5.4(c). With sharded retrieval, the root microservice has to wait until the responses from all shards return. In Parslo, we represent all the parallel shards as a single microservice node in the DAG, subject to a single partial SLO, since they operate concurrently. However, the latency distribution of the entire microservice node may differ from the distribution of a single shard, as the latency of the microservice node is determined by the slowest shard. In particular, when latency SLOs are defined based on tail latency, the microservice node representing all shards will have a higher tail latency than that measured at each instance. For example, if the probability of the observed latency being greater than or equal to a specific value is measured to be 1% at a single instance (i.e., the value represents the 99^{th} percentile tail latency), the probability of the observed latency being greater than or equal to the same value over the entire microservice node would be 1.99%, which represents the $\sim 98^{th}$ percentile tail latency of the microservice node.

The hockey-stick graph of a microservice node—which is used by the Parslo algorithm to calculate cost savings—represents the average or tail of the latency distribution at a given load. In Parslo, we seek to derive the hockey-stick graph for the entire microservice node based on the hockey-stick graph measured at a single shard. To measure the tail latency at a particular load, a large number of latency measurements need to be made to construct a latency distribution and determine its tail of interest. When a request goes through two parallel paths within a microservice node and the latency of the entire node is determined by the slowest path, as shown in Figure 5.4(c), the Probability Distribution Funciton (PDF) of the entire microservice can be inferred from the measured distributions at the two parallel paths using Equation 5.7, assuming the latencies of the two paths are independent. When a microservice performs indexing on multiple shards, the same procedure may be applied to incrementally derive the latency distribution of the entire microservice, based on the measured distribution of a single instance (shard). It is usually safe to assume the latency distributions of different shards are independent, given that, at a particular load, internal events within an instance—which are independent of the other instances—cause specific

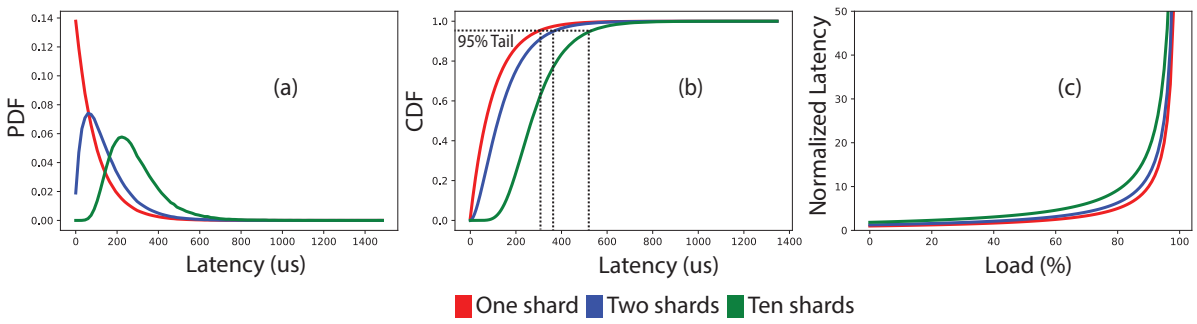


Figure 5.6: (a) PDF and (b) CDF resulting from the sharding transformation on the latency distribution of microservice exhibiting a 100μs mean latency with an exponential latency distribution, and (c) the transformed hockey-stick graph of a microservice with an M/M/1 queueing model.

requests to be slower than nominal ones and comprise the tail [139].

$$
\begin{aligned}
PDF_M(t) &= P(M=t) \\
&= P(A=t)P(B\leq t) + P(B=t)P(A\leq t) - P(A=t)P(B=t) \\
&= PDF_A(t)CDF_B(t) + PDF_B(t)CDF_A(t) - PDF_A(t)PDF_B(t)
\end{aligned}
\tag{5.7}
$$

Figure 5.6 illustrates the impact of inferring the PDF (a) and CDF (b) functions of a sharded microservice based on the latency distribution of a single instance, which is assumed to exhibit a 100μs mean latency with an exponential distribution. As these figures show, sharding causes the probability of small latencies to decreases while resulting in a higher probability for large latencies, since the overall microservice latency is detemined by the slowest shard. Figure 5.6(c) presents the resulting hockey-stick graphs for a sample microservice with an M/M/1 queueing system, for different numbers of shards. Parslo calculates the cost of sharded microservices based on such transformed hockey-stick graphs using Equation 5.7.

5.3.3 Branching Paths

In real cloud applications, such as the ones depicted in Figure 5.1, requests do not necessarily go through a single chain of microservices, but rather may branch through multiple paths, as shown in Figure 5.4(c). Branching makes partial SLO allocation more complicated. Parslo addresses this issue via a combination of mechanisms. First, Parslo seeks to equalize the SLO across all paths—by ensuring all possible paths within the service conform to the same average or tail latency SLO, Parslo ensures that all requests meet the end-to-end latency SLO, regardless of the path they take. To ensure it is possible to equalize all paths, Parslo only supports a particular class of DAGs, known as Nested Fork-Join (NFJ) DAGs.

Prior work has shown that checking whether a DAG is NFJ and, if not, converting it into an NFJ DAG through node replication can be performed in linear time [77, 57]. Replicating a node in a microservice DAG simply means that different instances of the microservice are subject to different latency SLOs. For example, Figure 5.7(a) depicts a non-NFJ DAG that has been converted to an NFJ DAG in Figure 5.7(b) by replicating node D into nodes D1 and D2. This replication means that D1 and D2 are treated as two separate microservice deployments, each with their own partial SLO that is enforced by their own auto-scaler. Despite the fact that the instances of D1 and D2 are effectively running the same code, they might be offered different load and target different latency SLOs.

During initialization, Parslo ensures that all branches of a fork are allocated an equal total budget of the initial SLO, by dividing the residual SLO from the branch with the largest number of nodes among all the nodes in each other branch, as shown in Figure 5.7(b). Then, Parslo obtains all combinations of nodes across branches by performing a Cartesian product across branches at each fork. Each combination of nodes represents an option for allocating a marginal SLO unit to all branches of a fork to maintain an equal SLO budget across all branches.

During the iterative optimization process, the algorithm selects the best candidate to

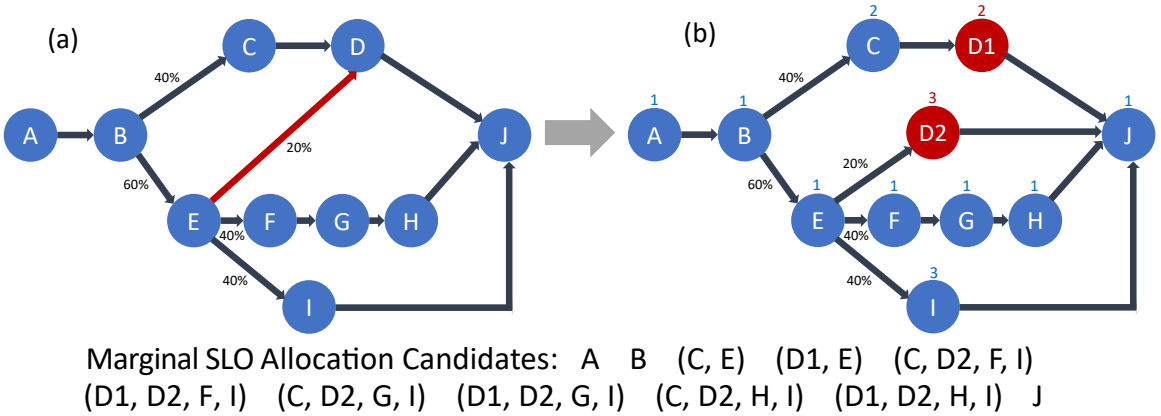


Figure 5.7: (a) an arbitrary DAG of microservices, (b) converting the DAG into an NFJ DAG; initial SLO allocations denoted on top of each node.

allocate the marginal SLO unit to achieve highest cost savings. When microservice DAGs include branches, candidates may comprise single nodes or combinations of nodes, wherein a marginal SLO unit is allocated to all nodes within a combination. Figure 5.7(b) lists all potential candidates for allocating marginal SLO in each iteration. With branches, the total cost of the entire service deployment depends upon the probability that each request might take a particular branch at a fork—the number of instances of a node is linearly proportional to its request arrival rate. As a result, Parslo also requires that DAGs be annotated with an estimated (or profiled) fraction of requests taking each branch at a fork. Equation 5.4 calculates the total cost for each node. However, with branches, the cost of each node is weighted by the fraction of requests passing through the node. For example, in Figure 5.7(b) the cost of node I is calculated by multiplying the cost calculated via Equation 5.4 to 0.24 (0.4×0.6). The weights for the nodes in combination (C, D2, F, I) would be (0.4, 0.12, 0.24, 0.24) whose sum is equal to 1.0.

The number of candidates Parslo must examine at each iteration may grow combinatorially with the branching degree of forks. But, since real microservice DAGs do not usually have branching degrees larger than 5, the number of combinations remains tractable. For example, whereas the DAG in Figure 5.7(b) has 11 nodes in total, the number of candidates Parslo must examine at an iteration is also 11.

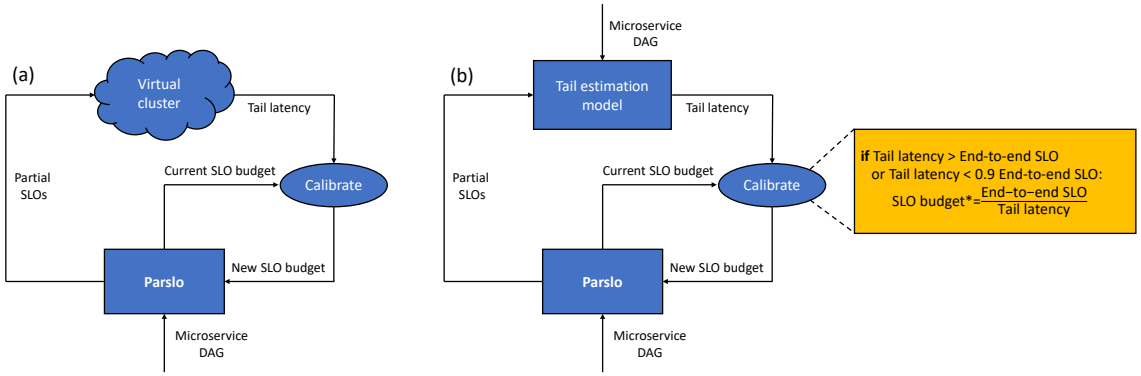


Figure 5.8: Parslo's (a) online and (b) offline SLO budget calibration framework.

5.4 Parslo: Calibration

In the previous section, we described Parslo's partial SLO allocation procedure, given an end-to-end latency SLO budget. The procedure described in the previous section only results in optimal SLO allocation if the latency SLOs are defined based on metrics, such as average latency, which exhibit an additive property. However, this approach would be highly conservative and wasteful if SLOs are defined based on metrics like tail latency, which do not exhibit such an additive property. As an example, with a chain of two microservices A and B, the procedure would allocate partial SLOs to them in such a way that the sum of SLO_A and SLO_B would be equal the end-to-end SLO budget. However, the end-to-end tail latency may be much smaller than the sum of the tail latencies of A and B, since the likelihood of a request experiencing exceptionally high latencies in both is very low, unless the two microservices are inter-dependent. Therefore, even if the allocated partial SLOs sum up to a value that is higher than the end-to-end SLO, the resulting tail latency may still meet the end-to-end SLO.

To address this issue, we augment Parslo with an iterative "calibration" approach—illustrated in Figure 5.8—wherein Parslo starts with an SLO budget which is equal to the end-to-end latency SLO, and iteratively increases the SLO budget until the resulting end-to-end tail latency is very close to the end-to-end latency SLO. The calibration framework modifies the SLO budget proportional to the ratio of the end-to-end tail latency and the

end-to-end latency SLO. Every time the SLO budget is modified, the Parslo SLO allocation algorithm is performed again to allocate all microservice-level partial SLOs and derive the resulting end-to-end latency distribution, and thus the end-to-end tail latency. If the resulted tail latency is higher or significantly (i.e., more than 10%) lower than the end-to-end latency SLO, the SLO budget will be re-adjusted, until the achieved the tail latency is only within 10% of the end-to-end latency SLO. We propose an online and an offline variant for the calibration approach, illustrated in Figures 5.8(a) and (b), respectively.

$$\begin{aligned} PDF_S &= PDF_A * PDF_B \\ &= P(S=t) = \int_{-\infty}^{\infty} P(A=x)P(B=t-x)dx \end{aligned} \tag{5.8}$$

$$PDF_S = P(S=t) = P(A=t)P(A) + P(B=t)P(B) \tag{5.9}$$

In the online approach, shown in Figure 5.8(a), the end-to-end tail latency must be measured at the virtual cluster every time the SLO budget is re-adjusted. Whereas we find that calibrating the SLO budget usually takes less than 10 iterations, this may still be costly and time-consuming, since each iteration involves (1) running the Parslo SLO allocation to calculate the optimal partial SLOs for the given SLO budget, (2) waiting for the corresponding auto-scalers to upsize/downsize the cluster accordingly, and (3) measuring a large number of end-to-end request latencies to collect high-confidence latency distributions. As a result, we recommend calibrating the SLO budget offline using the estimates of the latency distributions first and only perform an online calibration phase at deployment time, if necessary (i.e., if the resulting end-to-end tail latency is higher or significantly lower than the end-to-end latency SLO).

5.4.1 Offline Tail Estimation Model

We propose an offline tail estimation model to enable offline calibration of Parslo's SLO budget, as shown in Figure 5.8(b). The model estimates the end-to-end tail latency given a set of partial SLOs for all microservice nodes. After allocating a partial SLO to a microservice node, the maximum load each instance of the microservice can operate at will be known, which corresponds to a measured latency distribution, as shown in Figure 5.9. Assuming all microservice nodes are independent, we can combine these distributions to derive the end-to-end latency distribution for the entire microservice DAG, to estimate the end-to-end tail latency. Whereas different microservice nodes might be inter-dependent, we make such an independence assumption in our offline calibration phase, since it would be automatically corrected in the online calibration phase, if needed.

Our tail estimation model uses equations 5.8 and 5.9 to iteratively combine the latency distributions of different microservice nodes, until all distributions are reduced into a single end-to-end latency distribution. When two independent microservice nodes are chained, their latency distributions may be combined using a convolution operation, as shown in Figure 5.9 and Equation 5.8, to derive to the end-to-end latency distribution. Convolutions are used to derive the distribution of a sum of two random variables by considering all combinations of values from the two distributions that yield a particular summation. Figure 5.9 illustrates two microservices with the same behavior, which exhibit exponential latency distributions. As shown in the figure, while the tail latency for one microservice is $\sim 400\mu$s, the tail latency for the convolution of the two latency distributions—which represents chaining the two microservices—would be $\sim 600\mu$s, rather than 800μs. This is due to the lack of additive property for tail latency, which we explained earlier.

Similarly, Equation 5.9 may be used to derive the end-to-end latency distribution over a branching path (i.e., fork operator), shown in Figure 5.4(d). Our tail estimation model recursively employs Equation 5.8 to reduce all nodes within a chain in a branch into a single node and then employs Equation 5.9 to reduce the nodes in all branches of a fork to a

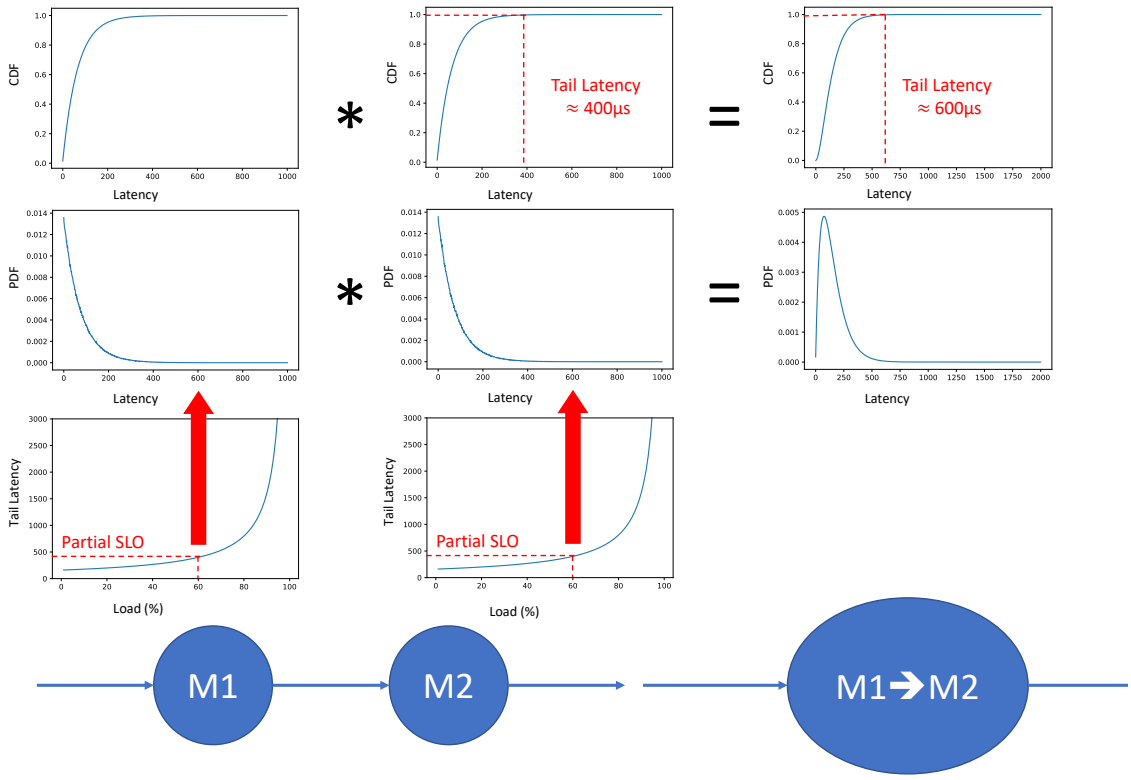


Figure 5.9: Combining latency distributions of two chained microservices using convolution in Parslo's offline tail estimation model.

single node. After all microservice nodes are reduced into a single node, the node's latency distribution may be used to obtain the end-to-end tail latency and compare it against the end-to-end latency SLO.

5.5 Evaluation

To evaluate Parslo, we employ stochastic queuing simulation, using the BigHouse Framework [134], based on the measured service time distributions of three common microservices from [141]: FLANN, McRouter, and Word Stemming (WS). We consider $M/G/k$ queuing systems based on these microservices running atop instance types with 8 or 16 processors with instance costs proportional to the number of processors. These assumptions are in line with VM offerings from all major cloud providers. Parslo is proven to achieve the optimal solution. However, to determine Parslo's improvements over the state-of-the-art, we compare our deployment costs to those achieved by the SLO allocation approach presented in GrandSLAm [95]. We consider a variant of GrandSLAm that allocates SLOs based on zero-load latency (i.e., average or tail of the service time distribution depending on the SLO definition) rather than the average service time, as we find this variant to always perform better (see Section 5.2.3). To evaluate real-world microservice-based applications, we apply Parslo to two microservice DAGS for the social network and media service from [59]. We consider simplified variants of these DAGs with small modifications as the original DAGs exhibit synchronous dependencies which Parslo does not suport.

5.5.1 Chains of Microservices

To determine the impact of different factors on the improvements of Parslo over GrandSLAm, we first consider chains of only two microservices. Figure 5.10 reports the cost ratio of Parslo over GrandSLAm, under (a) a relaxed SLO latency target, which is $10\times$ the sum of the zero-load latencies, and (b) a tight SLO latency target, which is $3\times$ the sum of the zero-load latencies, when SLOs are defined based on the average and 99^{th} percentile latency

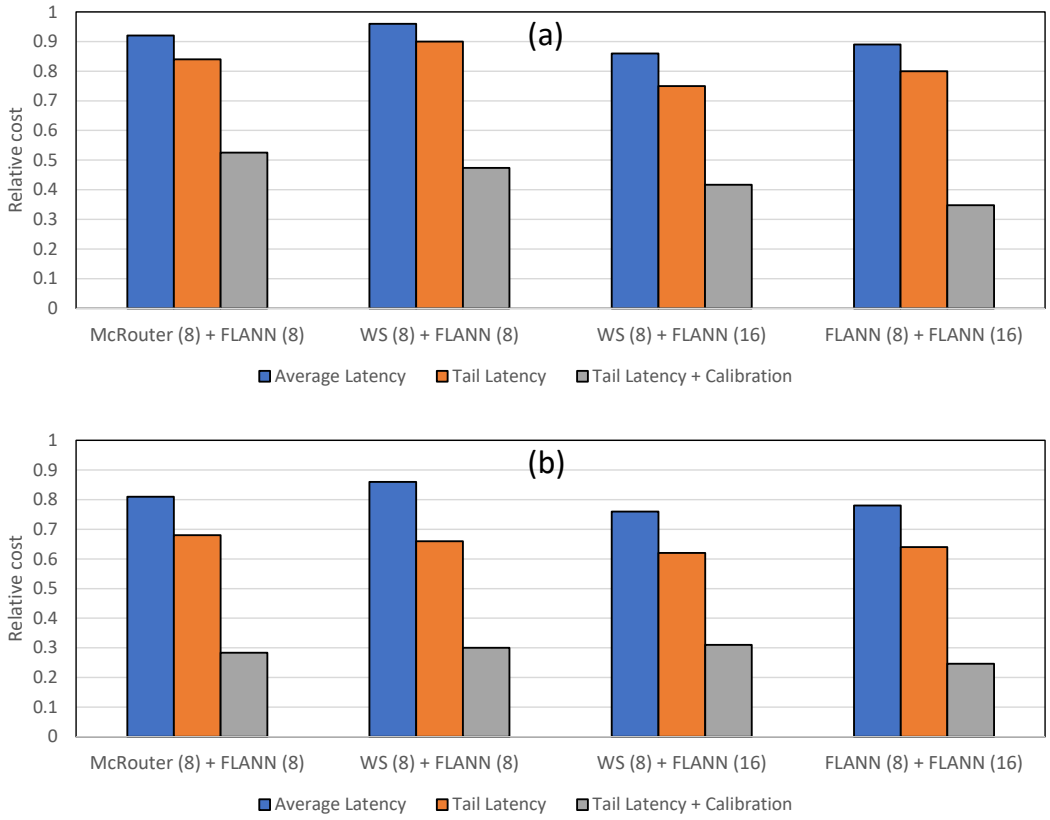


Figure 5.10: Relative deployment costs of chains of two microservices with different instance sizes and costs ahcieved by Parslo, compared to GrandSLAm when SLOs are defind based on the average and the 99^{th} percentile tail latency for SLO of (a) $10\times$ and (b) $3\times$ the sum of zero-load latencies.

(with and without calibration). As shown in the Figure, Parslo achieves larger improvements over grandSLAm when (1) SLOs are defined based on tail (rather than average) latency, (2) instance types are different (rather than the same), and (3) the SLO latency target is tight. Conclusion (1) is because, as explained in Section 5.2.3, GrandSLAm's approach is closer to optimal when zero-load latency is inversely proportional to maximum load, which is usually true when hockey-stack graphs represent average latency. In such scenarios, Parslo would have a smaller opportunity for improvement over GrandSLAm. Conclusion (2) is because GrandSLAm's approach is agnostic to instance costs and only seeks to minimize the number of instances. Conclusion (3) is because increasing the latency SLO results in much sharper increase in maximum utilization when latency SLOs are tight. Therefore, optimal SLO allocation is more critical in these regimes.

Finally, Parslo's improvements are higher when the two microservices are McRouter+FLANN compared to when they are WS+FLANN, because both WS and FLANN exhibit high-disparity service distributions. Hence, their hockey-stick graphs are more similar to one another, making GrandSLAm perform better and leaving smaller improvement opportunity for Parslo. As Figure 5.10 shows, calibration reduces the total deployment cost achieved by Parslo by $2\times - 3\times$, by addressing the lack of additive property in tail latency, which the non-calibrated variant of Parslo does not consider.

5.5.2 DAGs of Microservices

To demonstrate the key benefit of Parslo's mechanism to allocate partial SLOs to microservice DAGs, we consider 5 synthetic DAGs as well as the two DAGs for the social network and media service from [59], depicted in Figure 5.1. All DAGs exhibit parallel indexing as well as branching paths. Since GrandSLAm does not natively support DAGs, we consider a modified variant of it, wherein the end-to-end SLO is divided across the nodes within the critical path of the DAG proportional to the zero-load latencies. The critical path is the path with the largest sum of zero-load latencies. Non-critical paths are allocated SLO

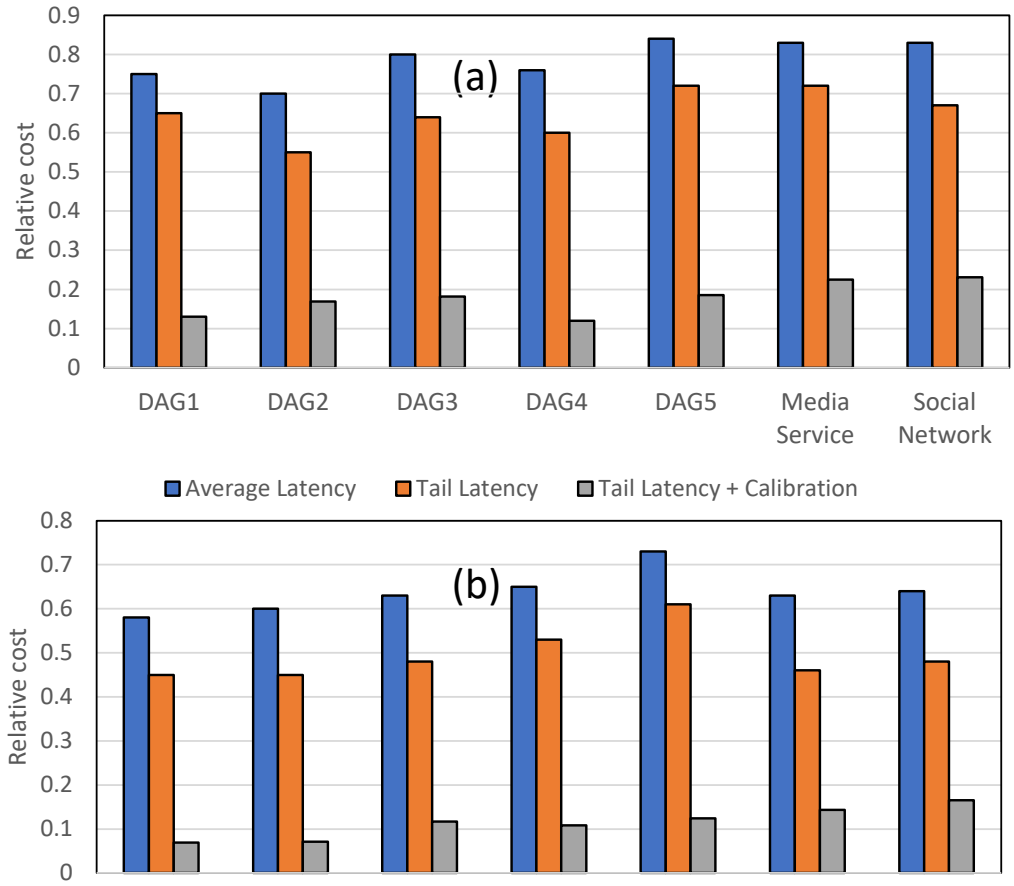


Figure 5.11: Relative deployment costs of microservice DAGs achieved by Parslo, compared to the DAG-aware modified variant of GrandSLAm when SLOs are defind based on the average and the 99^{th} percentile tail latency when SLO is (a) $3\times$ and (b) $10\times$the sum of the zero-load latencies on the critical path.

budgets equal to the allocation of the critical path and divide their SLO budget across nodes proportional to the zero-load latencies of the nodes on those paths.

Figure 5.11 reports the cost ratio of Parslo over the modified variant of GrandSLAm for DAGs, under a SLO latency target that is (a) $10\times$ and (b) $3\times$ the sum of the zero-load latencies on the critical path, when SLOs are defined based on the average and 99^{th} percentile latency (with and without calibration). All synthetic DAGs consider mixes of McRouter, WS, and FLANN workloads. Parslo's improvements are much larger in DAGs because GrandSLAm does not natively support fan-out and branching effects. Similar to chains the case of chains, Parslo's improvements are much larger when with tighter SLOs and

when latency SLO is defined based on tail, rather than average latency. All synthetic DAGs are calibrated offline as their nodes are, by definition, independent. In case of the two real DAGs, however, we perform final calibration to correct the potential errors of the offline calibration phase. As shown in Figure 5.11, total cost savings for the two real DAGs are a little smaller than synthetic DAGs due to dependencies across different nodes. Altogether, Parslo achieves up to more than $2\times/6\times$ (with/without calibration) reduction in deployment costs, compared to GrandSLAm, for the social network and media service DAGs.

5.6 Conclusion

We proposed Parslo as a Gradient Descent-based approach to allocate partial SLOs among nodes in a microservice DAG, enabling independent auto-scaling of individual microservices. Parslo employs novel mechanisms to be applicable to general DAGs, with microservice dependencies, parallel fan-out, and branching paths. Parslo achieves the optimal solution, minimizing the total cost for the entire service deployment. Our evaluation demonstrates that Parslo reduces service deployment costs by more than $6\times$ in microservice-based applications, compared to a state-of-the-art SLO allocation technique.

CHAPTER VI

μSteal: Preemptive Work and Resource Stealing for Mixed-Criticality Microservices

6.1 Introduction

Modern internet services are shifting away from single-binary, monolithic services into various loosely-coupled microservices, to enable rapid development, release, and frequent updates of cloud software [58, 59, 186, 183]. Microservice–based services are implemented as a Directed Acyclic Graph (DAG) composed of tens to hundreds of individual microservices, wherein each microservice node of the DAG is independently deployed and scaled. Microservice architectures have been adopted by major cloud-based companies, such as Facebook, Netflix, and Linkedin, as they significantly improve programmability, reliability, manageability, and scalability. For example, a Facebook news feed query may flow through a chain of microservices, such as Sigma (a spam filter), McRouter (a protocol router), Tao (a distributed social graph data store), and MyRocks (a user database) [183].

Since microservices arise from decomposing complex services into simpler components, common microservices are often found across multiple end-to-end services. For example, Facebook incorporates the same face detection/recognition and image understanding models into many user-facing services [76, 209]. Similarly, speech recognition and many other text and language understanding microservices may be common across services like web search,

translation, or digital assistants. However, given the dissimilarities in the orchestration and the number of microservices in the DAGs of different services, as well as varying end-to-end latency objectives, these common microservices may need to operate under differing latency constraints when deployed as part of each service.

The common and straight-forward solution is to deploy a dedicated instance pool (i.e., virtual machines or containers) for each end-to-end use case of a microservice [201]. These dedicated pools can be tuned for differentiated latency constraints and scaled appropriately for each end-to-end service deployment. However, in this chapter, we argue that dedicated pools can be wasteful; sharing an instance pool across multiple use cases for the same microservice can result in significant reduction in the total number of instances (and, correspondingly, compute and memory resources), especially if the latency constraints imposed by the service DAGs are highly asymmetric. We call microservice deployments with diverse latency requirements across classes of requests "mixed-criticality" microservices, and seek to facilitate their implementation.

Whereas sharing microservice instances across multiple deployments can result in remarkable resource savings, we show that it is only beneficial if the arriving requests from each criticality class (i.e., service deployment) are intelligently scheduled across the execution resources within an instance, to account for the varying latency requirements of each class—naively interleaving requests among classes leads either to over-provisioning or missed deadlines for requests with tighter requirements. In contrast, strict prioritization by latency constraint tends to starve requests in the most relaxed class. We examine multiple scheduling policies and show that no trivial policy achieves a competitive request throughput without violating latency requirements in any request class.

To address this challenge, we propose a request scheduling scheme, called μ*Steal*, that leverages preemptive work and resource stealing to schedule arriving requests onto processing cores within a mixed-criticality microservice instance. μSteal provisions differing "core reservations" for each request class, depending on their latency constraints, but allows a

class to steal cores from other reservations if the cores would otherwise remain idle. Still, when offered load within a class requires its full reservation, μSteal preempts stolen cores, returning them to their reserved class. By synergistically employing *core partitioning*, *work stealing*, and *preemption*, μSteal seeks to maximize the total throughput supported by an instance while ensuring all request classes meet their latency constraints.

μSteal's effectiveness depends critically on the optimal allocation of core reservations to request classes. μSteal allocates a higher processing capacity to a class by reserving more cores to it—poor allocation results in suboptimal throughput and efficiency. μSteal employs a feedback controller to tune reservations at runtime, since the arrival rate in each class may change over time. However, in the presence of load spikes, wherein a class's arrival rate increases suddenly, a feedback controller approach may take long to converge iteratively to an optimal core reservation distribution across request classes. Hence, we propose a queuing theory-based analytical approach to estimate the required core reservation per class, to minimize feedback controller convergence time.

To our knowledge, μSteal is the first scheduling framework for mixed-criticality microservices with varying latency requirements across request classes. Our real-system implementation shows that μSteal reduces the required number instances by up to $1.29\times$ compared to a deployment with dedicated instances for each request class and significantly outperforms any other scheduling scheme when using shared instances in a mixed-criticality microservice.

6.2 Background and Motivation

6.2.1 A case for mixed-criticality microservices

The same microservices may often appear in the DAG of multiple end-to-end services. As an example, Figure 6.1 depicts a scenario where a provider offers web search, translation, and digital assistant end-to-end services, and it accepts both text and voice inputs for search

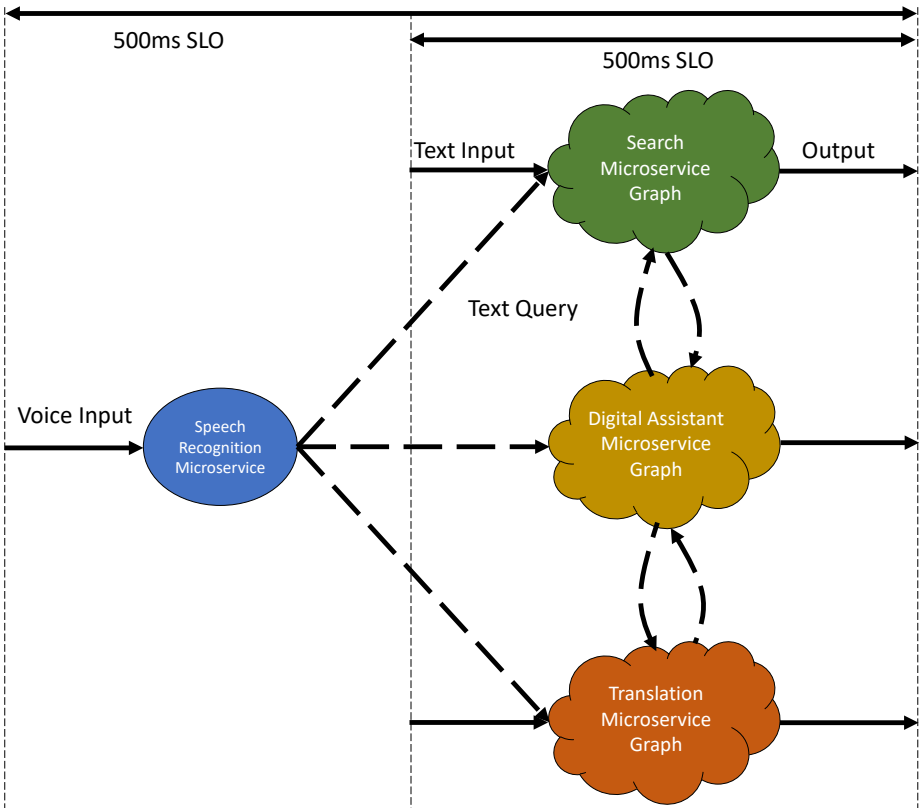


Figure 6.1: Speech recognition as a shared mixed-criticality microservice.

and translation. The digital assistant service may itself generate queries to the search or the translation services, depending on its input voice command. The speech recognition microservice is shared across the DAG of all three end-to-end services.

Assuming the end-to-end latency SLO for all of these services is 500*ms* to achieve a desirable user experience, the speech recognition microservice must satisfy different *parial SLOs* depending on the end-to-end service class. For example, suppose the average time taken to complete a search query is significantly longer than the average time taken to finish a translation query. In this case, the latency SLO for the speech recognition microservice in the web search DAG must be significantly smaller than the SLO for the same microservice in the translation DAG. On the other hand, for all microservices in the search or translation DAGs, the latency SLO must be tighter if the query originates from voice (rather than text) input, given that a fraction of the 500*ms* SLO is spent in the speech recognition microservice. Similarly, all microservices in the search or translation DAGs must meet significantly

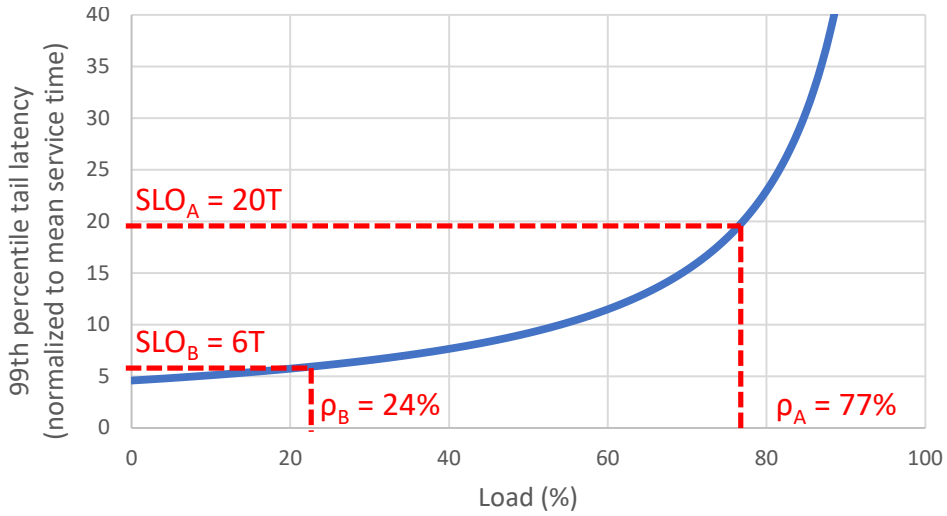


Figure 6.2: Maximum load supported by a microservice instance under different latency SLOs.

tighter SLOs if their input query is generated by the digital assistant service, rather than directly from a user. We call microservices with multiple latency SLOs "mixed-criticality" microservices; a term we borrow from the embedded/real-time systems community [20].

6.2.2 Sharing instances

Figure 6.2 illustrates a synthetic load vs. latency profile for one instance of a microservice, with an $M/M/1$ queuing model. Load refers to the ratio of the request arrival rate (λ) to a single instance, by the total service rate of the instance ($c\mu$ where c is the total number of cores in an instance and μ is the maximum service rate of a core). As shown in the figure, as load increases, the response time increases by a semi-exponential rate—especially at loads above $\sim 50\%$—due to the increased queuing delay in the instance's request queue. When the microservice's latency SLO is relaxed to a higher value, each instance may operate at higher utilization while still meeting the SLO, allowing the auto-scaler to reduce the number of microservice instances.

Consider a microservice with the load-latency profile shown in Figure 6.3 and arriving requests belonging to two classes A and B. Whereas A has a relaxed SLO that requires tail latency to be bounded within $20\times$ the mean service time, B has a strict SLO that requires

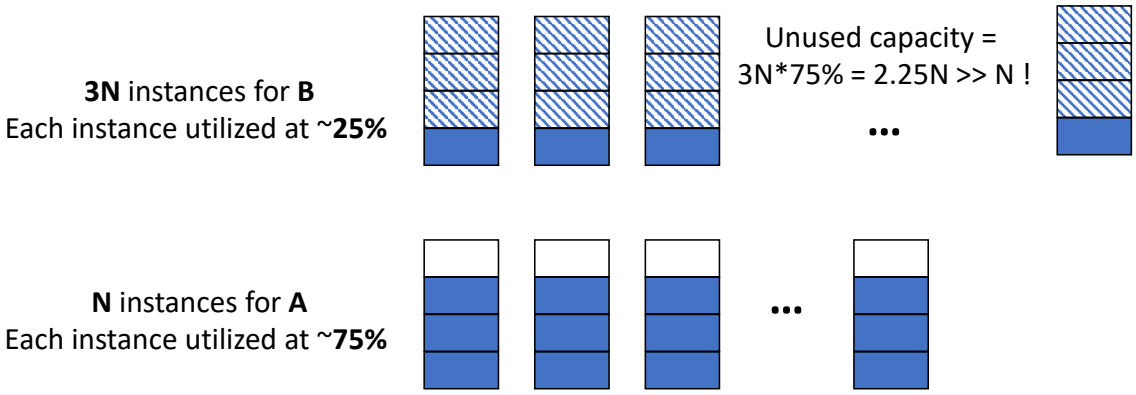


Figure 6.3: An illustrative example showing the resource saving opportunity from sharing instances across different deployments of a microservice with different latency SLOs.

the tail latency to be bounded within only $6\times$ the mean service time. As a result, if we deploy separate instances to service requests for A and B classes, whereas each instance deployed for A can support at most 77% utilization, each instance deployed for B is limited to only 24% utilization without violating their corresponding latency SLO. To simplify the calculations, we assume each instance deployed for A can be utilized up to 75% and each instance deployed for B can be utilized up to 25%. Hence, as shown in Figure 6.3, assuming equal arrival rates for A and B, if A needs N instances to meet its latency SLO, B needs $3N$ instances (each instance for A can support $3\times$ higher utilization).

As shown in Figure 6.3, the underutilized B-instance capacity equals $0.75 * 3N = 2.25N$ (each of the $3N$ instances is underutilized 75% of the time), which is more than twice the total capacity needed by A to meet its latency SLO. This example shows that deploying distinct instance pools for each request class results in myriad wasted resources. While sharing the instances across request classes may reduce instance count by up to 25% in this example, further resource savings are possible if the classes' arrival rates are also asymmetric (in addition to their latency SLOs). However, naively sharing the instances across multiple request classes may result in significant SLO violations—since B needs exactly $3N$ instances to meet its latency SLO in our example, any interference caused by requests from A will result in SLO violations for B.

6.2.3 Scheduling within an instance

Task-parallel frameworks, such as Cilk [16], usually employ per-core local task queues to observe the dependencies and maximize locality across related tasks while minimizing synchronization overheads. To maximize throughput, these frameworks often employ *work stealing*, wherein a core usually processes tasks from its own local queue, but, if its local queue becomes empty, the core steals a task from another non-empty queue. When all tasks constitute a single job, the overall job execution latency translates roughly to task processing throughput. Hence, a work-stealing task scheduler seeks to maximize task throughput to reduce job execution latency.

However, in cloud microservices, there is no dependency or locality among requests. Furthermore, the metric of interest in cloud microservices is the response time for individual requests. As a result, unlike task-parallel frameworks, cloud microservices usually employ a single shared request queue across cores, to observe the FIFO arrival order of the requests and prevent Head-of-Line (HoL) blocking caused by requests with exceptionally long service times, minimizing the response time distribution mean and tails [139]. Microservices exhibit mean service times ranging from tens of microseconds to single-digit seconds [183]. Whereas implementing a shared queue across cores can be challenging for short μs-scale microservices, recent frameworks have sought to mitigate its overheads in software [91] or hardware [36].

We envision a baseline scheduler for mixed-criticality microservices, shown in Figure 6.4(a), wherein requests from all classes are spread over all instances. So, there are multiple request queues for different classes within each instance. An ideal request scheduler must provide some form of prioritization for the request classes with tighter latency SLOs or higher request arrival rates. Unfortunately, as we will show in Section 6.6, strict prioritization of one class over another results in starvation and drastic SLO violations for under-prioritized classes. In contrast, partitioning the cores across classes results in wasted resources, requiring more instances to be launched, since the resulting scheduler is no longer

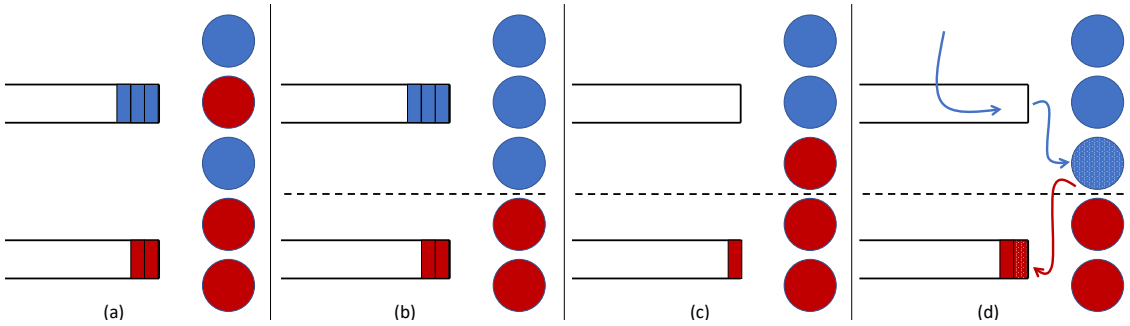


Figure 6.4: (a) A baseline mixed-criticality microservice deployment with multiple request queues belonging to different classes shared across all cores, (b) partitioning the cores across request classes, (c) partitioning augmented by work/resource stealing, (d) preempting the youngest request from the stealing class as performed by the μSteal scheduler. Note that μSteal allocates a core reservation "count" to each class, rather than a fixed set of cores, as shown in the figure.

"work-conserving"—cores allocated to one class may remain idle if the corresponding request queue is empty while there are requests pending in another class's queue.

Earliest Deadline First (EDF) is a classic scheduling algorithm for real-time systems where individual tasks have a deadline. It can be shown that if a collection of tasks with different arrival times and deadlines are schedulable by any algorithm such that all tasks complete by their deadline, EDF will schedule this collection of tasks so they all complete by their deadline [219]. It has further been shown that EDF minimizes the fraction of reneged work—the residual work lost due to elapsed deadlines—under heavy traffic [108].

Prior work [95] suggests setting the SLO latency target for each request in a microservice as its deadline and scheduling requests using EDF, resulting in a near-minimal SLO violation rate. The term near-minimal is used as EDF guarantees to minimize the the total reneged work, rather the the total number requests that miss their latency target. In microservices with a single latency SLO, EDF simply functions as a First Come First Serve (FCFS) scheduler, as the deadline (i.e., latency SLO) for all requests is the same. GrandSLAm [95] suggests a particular variant of EDF for microservice environments, called *Least Slack First (LSF)*, which re-orders requests to make up for the slowdowns in the previous microservice stages, seeking to minimize end-to-end SLO violations. LSF re-orders requests at each microservice stage according to the original arrival time of the requests (at the entire end-to-end service),

rather then their arrival time to the individual microservice node.

Whereas EDF/LSF achieves a near-minimal SLO violation rate for single-SLO microservices, in case of mixed-criticality multi-SLO microservices, it seeks to minimize the total SLO violation rate across all requests for each class—EDF/LSF is unaware of request classes and treats all SLO violations the same regardless of class. As such, the algorithm will tend to prioritize requests from one class over another, especially if the classes exhibit asymmetric request arrival rates. Notably, in such cases EDF/LSF favors classes with the higher request arrival rates, as doing so minimizes the overall SLO violation rate.

Our goal in this chapter is to design a scheduling scheme for mixed-criticality microservices, which—unlike EDF/LSF—is aware of the request classes and seeks to maximize the total request throughput within each instance (thereby reducing the number of instances) while ensuring each request class meets its latency SLO.

6.3 The μSteal Framework

In this section, we present the μSteal framework, a runtime request scheduling system to be deployed at each instance of a mixed-criticality microservice. μSteal schedules incoming requests onto cores so as to maximize the total request processing throughput per instance—thereby minimizing the total number of instances—while ensuring all request classes meet their latency SLO. μSteal framework considers multiple request classes each comprising a separate request queue (shared across all cores) and a pre-defined latency SLO, as depicted in Figure 6.4. Typically, microservice runtime frameworks follow variants of FCFS scheduling to minimize queuing delay and tail latency. Hence, μSteal also seeks to adhere to FCFS scheduling within each request class and only makes decisions on the interleaving of request admissions across classes. As a result, μSteal's main responsibility is to decide from which non-empty queue a core should select a request when the core becomes idle (i.e., finishes processing the previous request). μSteal employs a core partitioning scheme augmented with a preemptive work/resource stealing mechanism, to account for the asymmetric latency

SLOs and arrival rates of different request classes, while providing a "work conserving" scheduler to achieve maximum throughput. That said, note that μSteal can implement any scheduling policy—such as EDF/LSF—within a request class and is composable with other microservice request schedulers [91, 95], enabling them to perform well for mixed-criticality multi-SLO microservices.

6.3.1 Stealing-enabled scheduler

Algorithm 1 presents the high-level approach by which μSteal steers requests to cores. The algorithm considers N cores and M classes of requests with different latency SLOs. The algorithm allocates each request class a different "core reservation", wherein the reservations for all classes add up to the total number of cores (i.e., $r_1 + r_2 + ... + r_M = N$). The core reservation for a request class represents the number of cores a class is guaranteed to be allocated if there are sufficient available requests in the class's corresponding queue to utilize the allocated cores. This approach is conceptually equivalent to dividing the cores across request classes by the same fraction, as shown in Figure 6.4(b). Core reservations are the mechanism that the μSteal scheduler employs to partially prioritize one class over others—when a class needs a higher processing capacity (i.e., exhibits a significantly stricter latency SLO or a higher arrival rate), it is assigned a larger core reservation, ensuring it can claim cores when it needs them.

Since the framework seeks to maximize request throughput, it implements a work conserving scheduler, wherein no core remains idle if there is at least one request pending in any request queue. Hence, when a request arrives, if there is any idle core available within the instance, the request is dispatched to the core for processing. This may result in a class using cores in excess of its reservation (i.e., "stealing cores" from another class). Similarly, in the equivalent view of allocating a fixed share of cores among classes, this effect can be interpreted as a class with idle cores "stealing work" from a different class with pending requests and no idle cores, as shown in Figure 6.4(c).

Algorithm 3: μSteal Request Scheduling Procedure

```
1  event (core becomes available)
2      core.previous-request.class.decrement-allocated-cores()
3      best-class = NULL
4      lowest-core-reservation-ratio = inf
5      for class in all classes do
6          if !class.queue.empty() then
7              n = class.allocated-cores()
8              r = class.reservation()
9              If n/r < lowest-core-reservation-ratio
10             lowest-core-reservation-ratio = n/r
11             best-class = class
12         end
13     end
14     if best-class!=NULL then
15         request = best-class.queue.pop-front()
16         core.process (request)
17         request.class.increment-allocated-cores()
18     end
19     else
20         idle-cores.push-back(core)
21     end
22 event (request arrives)
23     if !idle-cores.empty() then
24         core = idle-cores.pop-front()
25         core.process (request)
26         request.class.increment-allocated-cores()
27     end
28     else
29         n = request.class.allocated-cores()
30         r = request.class.reservation()
31         if n<r then
32             best-class = NULL
33             highest-core-reservation-ratio = n/r
34             for class in all classes do
35                 n = class.allocated-cores()
36                 r = class.reservation()
37                 if n/r > highest-core-reservation-ratio then
38                     best-class = class
39                     highest-core-reservation-ratio = n/r
40                 end
41             end
42             victim = best-class.youngest-running-request()
43             core = victim.running-core()
44             core.preempt-running-request()
45             victim.class.decrement-allocated-cores()
46             victim.class.queue.push-front(victim)
47             core.process(request)
48             request.class.increment-allocated-cores()
49         end
50         else
51             request.class.queue.push-back(request)
52         end
53     end
```

In contrast, when a request arrives and no idle core is available within the instance, the framework checks whether the corresponding class is presently utilizing at least as many cores as its reservation. If present usage exceeds reservation, the request is enqueued to the corresponding request queue and waits to be processed in FIFO order within its queue. However, if the class's present usage is below its reservation, the framework preempts the youngest request from the class that is presently exceeding its reservation the most (i.e., by

the largest ratio), appends the preempted request to the head of its corresponding queue, and allocates the core to the newly arrived core, as shown in Figure 6.4(d). Said differently, if class i with r_i reservation is allocated n_i cores and class j with r_j reservation is allocated n_j cores, the framework compares $\frac{n_i}{r_i}$ and $\frac{n_j}{r_j}$ and preempts the youngest running request from the class with the highest ratio. By preempting the youngest such request and prepending it to the corresponding queue, the algorithm guarantees FIFO ordering within each class.

When a core becomes idle (i.e., finishes processing its previous request), it dequeues a new request from the class with a non-empty request queue and the lowest ratio of present usage to reservation (i.e., $\frac{n_i}{r_i}$). Because of the nature of the prioritization algorithm, a core will dequeue work from the same class as the just-completed request if that request class is not presently stealing from another class (there cannot be more than one non-empty queue with $\frac{n_i}{r_i} < 1$). As a result, while a class uses fewer cores than its reservation, the class is guaranteed to retain the same cores until its queue empties.

To summarize, the μSteal framework follows three key scheduling concepts: *resource partitioning*, *work stealing*, and *preemption*. By reserving cores to each request class, the framework conceptually partitions the cores among classes. However, work stealing allows request classes to steal idle cores beyond their reservation. Such work stealing ensures μSteal's scheduling is work conserving, wherein no cores idle if there is at least one request pending in any queue. Finally, the framework leverages preemption to ensure request classes always can seize as many cores as they are reserved, despite employing work-stealing.

6.3.2 Tuning reservations

The μSteal framework maximizes request throughput at every instance (thereby minimizing required instance count) while ensuring all request classes meet their latency SLOs by allocating asymmetric processing capacities among classes. μSteal allocates more processing capacity to a class by assigning it a larger core reservation—poor reservation allocation results in suboptimal throughput and efficiency. Unfortunately, optimal reservation allo-

cation depends upon the relative arrival rate among classes, which may change over time. Although significant load changes are infrequent, small changes in request class arrival rate are common. Since different request classes are independent, their arrival rates do not change together. Hence, optimal core reservation allocation also changes over time.

To ensure the optimal reservation for each request class at all times, μSteal employs an iterative feedback controller similar to that used in various frameworks, such as PARTIES [27]. Whereas such frameworks allocate numerous resources (cores, memory, LLC, etc.) across multiple applications, μSteal only allocates core reservations across multiple request classes within a single application (i.e., microservice), and hence employs a simpler controller.

Algorithm 2 presents the iterative feedback-loop procedure the μSteal framework uses to tune core reservations upon changes in arrival rates. At every epoch, the framework checks for SLO latency target violations in each request class and re-balances core reservations if at least one class is violating its SLO (i.e., latency target violation above 1% for an SLO defined based on 99% tail latency) and at least one class is meeting its SLO. (If all classes are violating SLO, the system is overloaded and no feasible adjustment can bring the system back into SLO). To rebalance reservations, the framework shifts one core from the request class with the lowest latency target violation rate to the request class with the highest latency target violation rate. If reducing the reservation of the class with the lowest latency violation rate results in an SLO violation (i.e., latency target violation goes above 1%), the shift is reverted and instead a core is taken from the request class with the second lowest latency target violation rate, and so on.

The iterative procedure continues until either all request classes meet their SLO or no solution can be found—that is, taking a core from any request class that meets its SLO results in an SLO violation (i.e., latency target violation above 1%). In this case, the procedure informs the auto-scaler that it cannot meet its SLO and the auto-scaler must reduce the per-instance load by increasing the number of instances. If there is a way to distribute

Algorithm 4: Feedback Controller for Tuning Reservations

```
1  Repeat
2      reset-epoch-log()
3      lowest-violation-rate = inf
4      lowest-violation-class = NULL
5      highest-violation-rate = 0
6      highest-violation-class = NULL
7      for class in all classes do
8          if class.last-epoch-log.latency-violation-rate() > highest-violation-rate then
9              highest-violation-rate = violation-rate
10             highest-violation-class = class
11         end
12         if class.last-epoch-log.latency-violation-rate() < lowest-violation-rate and class.is-available() then
13             lowest-violation-rate = violation-rate
14             lowest-violation-class = class
15         end
16     end
17     if highest-violation-rate > ACCEPTED-VIOLATION-RATE then        // 1% for 99^th percentile tail
         latency SLO
18
19         if lowest-violation-class!=NULL and lowest-violation-rate < ACCEPTED-VIOLATION-RATE then
20             lowest-violation-class.decrement-reservations()
21             highest-violation-class.increment-reservations()
22             wait-for-an-epoch()
23             if lowest-violation-class.last-epoch-log.latency-violation-rate() > ACCEPTED-VIOLATION-RATE then
24                 lowest-violation-class.mark-unavailable()
25                 lowest-violation-class.increment-reservations()
26                 highest-violation-class.decrement-reservations()
27             end
28             else
29                 mark-all-classes-available()
30             end
31         end
32         else
33             alert-auto-scaler("overload")
34             mark-all-classes-available()
35         end
36     end
37     else
38         mark-all-classes-available()
39     end
```

core reservations across classes so that all request classes meet their SLO, this procedure is guaranteed to find a solution, as the configuration space is by definition convex for each pair of request classes *, and the algorithm iteratively searches over the entire configuration space.

6.4 Reservation Allocation During Load Spikes

As explained in the previous section, if there is a feasible reservation allocation to meet all SLOs, the iterative feedback-loop procedure presented in Algorithm 2 is guaranteed to find it. In steady load conditions, changes in the arrival rate of each class are small (i.e.,

*Taking a core reservation from a class and giving it to another class would, by definition, result in an increased SLO violation rate for the former and a reduced SLO violation rate for the latter class.

within at most 20% [10]). As a result, typically at most one to two cores need be shifted among a pair of classes. Furthermore, since the auto-scalers usually overprovision instance count in anticipation of load spikes, it is likely that the instance's reservation configuration need not change in response to small changes in arrival rates.

However, under a transient load spike, the arrival rate of a particular request class might drastically increase. In this case, the required core reservation for each class may differ significantly from the current configuration. Algorithm 2 might require tens of iterations to eventually converge to a new working configuration. During this search, queues may build in several request classes, leading to high tail latencies both during and after the load spike. Each reservation configuration the algorithm considers requires a rebalancing epoch (comprising hundreds of thousands of requests) to evaluate to ensure a statistically meaningful measurement of latency tails [221].

To address this issue, we propose an analytical approach to initialize the search for optimal core reservations upon load spikes to accelerate convergence of algorithm 2 to examine only a few configurations. We next describe the mathematical intuition behind our reservation initialization approach:

Square-root staffing rule. Our analytical approach for assigning core reservations relies on the Square-Root Staffing (SRS) rule. SRS, described in Equation 6.1, is a general capacity planing tool that provides a simple estimation of the required capacity to service a given load at Quality- and Efficiency-Driven (QED) conditions [72]. At a high-level, QED conditions describe scenarios wherein a service is allocated sufficient capacity to meet its Quality-of-Service (QoS) requirements (tail latency SLO in our case) but the allocated capacity is not significantly higher than the required capacity, to minimize the cost and prevent resources from being wasted. Although it is desirable that a microservice always operates under QED conditions, since auto-scalers over-provision instance count in anticipation of load spikes, the system does not operate in the QED regime in steady load conditions (due to the over-provisioning to guard against load increase). However, when

load spikes occur, the system enters the QED regime, making SRS a well-suited estimation tool for capacity planning in the presence of load spikes.

$$C = R + \beta\sqrt{R} = \frac{\lambda}{\mu} + \beta\sqrt{\frac{\lambda}{\mu}} \tag{6.1}$$

In Equation 6.1, C estimates the total capacity or the number of resources required for a service to meet its QoS requirements. R represents the number of resources needed to just service all the requests without infinite queuing, regardless of the QoS, which is equal to $\frac{\lambda}{\mu}$, wherein λ represents the total request arrival rate and μ represents the service rate of a single resource. β represents the QoS target. Whereas β can be mathematically calculated for an $M/M/k$ queuing system (with the QoS metric being the probability of queuing) for accurate capacity planing, it can also be used with any other QoS metrics as an estimation tool for capacity planning [72]. In such cases, β can be calculated based on the measured maximum achievable resource utilization for a given QoS target.

Since the QoS is defined based on the tail latency SLO in our case, we can profile an in instance to measure the maximum load it can service without violating the latency SLO. Suppose, for example, an instance has ten cores and can sustain up to 40% utilization. Then, the β parameter for the latency SLO is 3.0, given by solving the equation $10 = 4 + \beta\ \sqrt{4}$. In this example, R is set to four, as four cores are sufficient to service all requests without infinite queuing. Similarly, if a 10-core instance can achieve a maximum utilization of 90% without violating the SLO, the β is 0.33, given by solving the equation $10 = 9 + \beta\ \sqrt{9}$.

Allocating core reservations using SRS. To estimate the required core reservation for each request class, we first calculate the β_i parameter for each request class and then formulate ρ_i as the maximum load an instance can sustain without violating the SLO of class i, if it is allocated a reservation of r_i cores. Equation 6.2 formulates the required reservation for class i to operate at load ρ_i, with N representing the total number of cores and f_i representing the fraction of incoming requests belonging to class i. Equation 6.3, calculates K_i as the expected number of cores available for use by class i if the instance

operates at load ρ_i. This equation assumes that class i always has access to r_i cores, as this quantity of cores are reserved for class i, but only may access the remaining cores if they are not consumed by other classes.

$$C_i = R_i + \beta_i\sqrt{R_i} = N\rho_i f_i + \beta_i\sqrt{N\rho_i f_i} \tag{6.2}$$

$$K_i = r_i + (N - r_i)(i - \rho_i(1 - f_i)) \tag{6.3}$$

Equation 6.4 equalizes the number of cores class i needs to meet its SLO under load ρ_i and the expected number of cores it finds available when needed under load ρ_i with r_i reserved cores. The resulting quadratic equation is solved in Equation 6.5, assuming r_i is constant and ρ_i is variable. As a result, Equation 6.5 can be used to find the maximum load where class i is expected to meet its SLO if it reserves r_i cores. At loads below ρ_i, K_i is larger than C_i, indicating that class i has access to more cores than it needs to meet its SLO. In contrast, at loads higher than ρ_i, C_i exceeds K_i, indicating that class i needs more cores than are expected to be available.

By exploring different combinations of reservations across different classes and considering the maximum load all classes can support without violating their SLO for each configuration—formulated in Equation 6.6—we can estimate the best reservation configuration that maximizes the load at which all classes meet their SLO.

$$C_i = K_i \;\rightarrow\; \rho_i(r_i - N - r_i f_i) - \sqrt{\rho_i}(\beta_i\sqrt{N f_i}) + N = 0 \tag{6.4}$$

$$\rho_i = \sqrt{\frac{\beta_i\sqrt{N f_i} - \sqrt{\beta_i^2 N f_i - 4N(r_i - N - N f_i)}}{2(r_i - N - r_i f_i)}} \tag{6.5}$$

$$\rho = min(\rho_1, \rho_2, ..., \rho_i, ...) \tag{6.6}$$

As an example, Figure 6.5, considers two classes A and B with equal arrival rates (i.e., $f_A = f_B$), $\beta_A = 1$, $\beta_B = 2$, and ten total cores. The figure shows the maximum load each

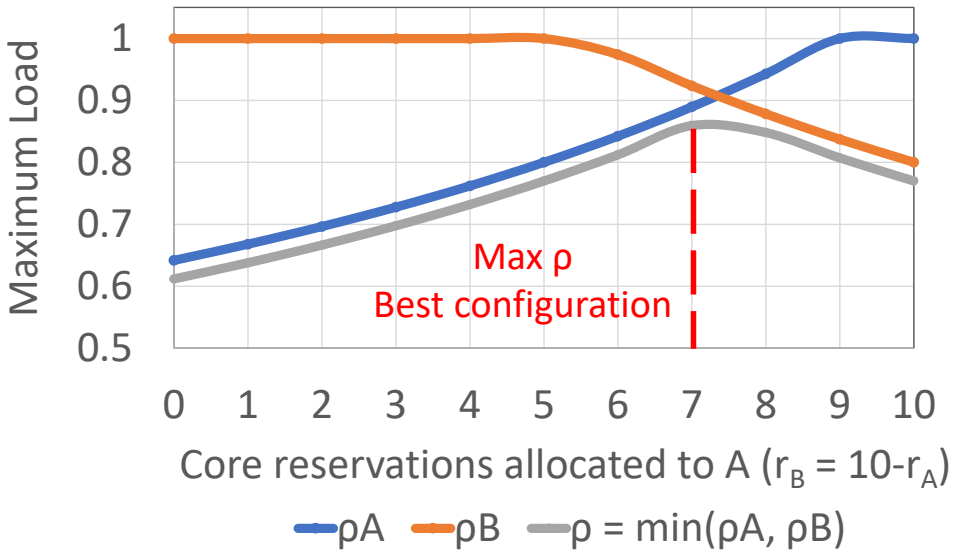


Figure 6.5: Maximum load supported by each class under different reservation configurations, estimated by μSteal's analytical reservation tuning tool.

class can support for each combination of reservations and the overall maximum sustainable load (i.e., the minimum of the two). As shown in the Figure, the maximum load is achieved when class *A* reserves 7 cores and class *B* reserves 3 cores. As illustrated by the example, Equation 6.6 may be used to estimate the maximum load given a reservation configuration without iterative search via Algorithm 2. In our experience, the configuration given by our analytical approach is at most 1-2 steps away from the optimal configuration found by Algorithm 2. When one of the classes experiences more than 20% arrival rate spike, μSteal uses Equation 6.6 to estimate the best reservation configuration first, before tuning it using Algorithm 2, to minimize convergence time.

6.5 Implementation and Methodology

We implement μSteal atop the Shinjuku [91] framework. Shinjuku is a user-level software data-plane that optimizes operating system threading and notification mechanisms to enable frequent, efficient preemption of μs-scale microservices. Shinjuku's main goal

is to reduce tail latency of high-disparity microservices via *processor sharing*, wherein pending requests are time multiplexed among cores to avoid HoL blocking and excessive queueing delays caused by rare, long tasks. It supports a single shared request queue across all cores and implements preemption at fixed time quanta, by allocating a separate thread to each request and provisioning a large thread pool to address cases with many queued requests. We modify Shunjuku to support multiple request queues corresponding to different request classes and perform preemptions when needed according to the μSteal scheduler (i.e., Algorithm 1).

We deploy our implementation of the μSteal scheduler on a 24-core Intel Xeon server, which represents a single instance of a particular microservice. We consider varying arrival traffic mixes and multiple latency SLOs for different request classes, and measure the maximum load our server can sustain without violating the latency SLO for any request class. The inverse of the maximum load corresponds proportionally to the required instance count. We consider a speech recognition microservice from the DjiNN and Tonic suite [75], which performs a neural network inference per request, and an Image search microservice from μsuite [185], which performs locality sensitive hashing. We compare μSteal against a baseline system where separate pools of instances are deployed for each request class, and cases where the same instances are shared across deployments and requests are scheduled via the following scheduling policies: FCFS, longest queue (idle core picks the next request from the longest queue), strict priority (always pick the next request from the class with the most strict latency SLO), EDF, static core partitioning, and static core partitioning augmented by work stealing. We consider preemptive variants for strict priority and EDF policies to perform a fair comparison again μSteal.

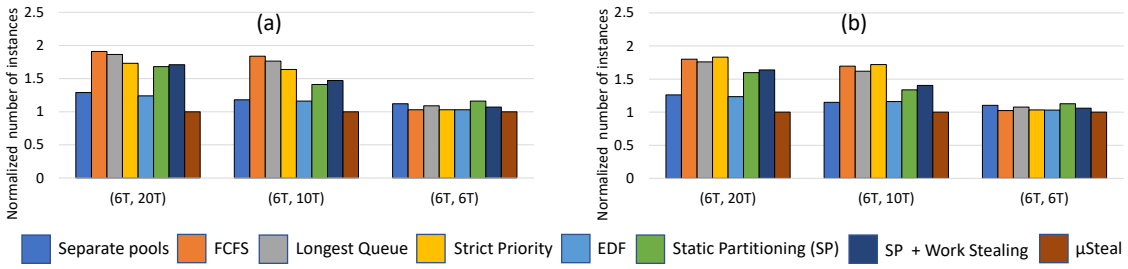


Figure 6.6: Normalized total number of instances for for (a) speech recognition and (b) image search microservices for deploying separate instances as well as sharing the instances across deployments with different scheduling policies. The arrival rates for both classes are equal and the latency SLOs are denoted in (6T, 6T) format, wherein 6T means that the 99^{th} percentile tail latency target for the SLO is equal to 6× mean service time.

6.6 Evaluation results

6.6.1 Symmetric traffic

Figure 6.6 reports the maximum required number of instances for the (a) speech recognition and (b) image search microservices for two request classes when each accounts for half of arriving requests. We consider different combinations of latency SLOs for the two classes. In all combinations, one class has a strict latency SLO (99^{th} percentile tail latency less than 6× the average service time, 6T). We consider three alternatives for the second class, respectively, 6T, 10T, and 20T latency SLOs.

As the figure shows, μSteal improves instance count as compared to dedicated instance pools per request class by up to 1.29× when the latency SLOs are highly asymmetric (6T, 20T) but the improvements shrink as the latency SLOs near one another. μSteal performs significantly better than all other scheduling policies when sharing instances across request classes. μSteal particularly outperforms FCFS by up to 1.93× as FCFS hurts the class with the stricter latency SLO by making its requests wait behind those of the other class. We conclude that naively sharing instance pools necessitates more instances than dedicated instance pools. Whereas prioritizing requests from the stricter SLO class seems to address FCFS's shortcomings, the strict priority policy only closes the gap to 1.73×; prioritizing the class with the tightest SLO leads to starvation and long wait times in the other class.

Static core partitioning also leads to a high instance count, 1.68× higher than that achieved by μSteal. Static partitioning results in a non–work-conserving scheduler, leaving cores from one class idle even when there are requests pending for the other class. Whereas augmenting static partitioning with work stealing seems to improve its performance, it actually increases the required count to 1.71× that required by μSteal. Work stealing, alone, leads to the cores assigned to the stricter SLO class to become occupied by requests from the other class. Thus, the requests in the stricter class miss their SLO. This result shows the importance of *preemptive* work stealing as envisioned in μSteal.

EDF is the only scheduling policy (other than μSteal) that improves instance count relative to dedicated instance pools. EDF naturally prioritizes for the class with the tighter SLO but also considers the more relaxed SLO class when the request at its head has spent enough time waiting. However, EDF instance count is still 1.24× higher than that required by μSteal, since, as previously noted, EDF is unaware of differentiated request classes, and seeks only to minimize the total number of SLO violations.

μSteal's improvements are minimal when both classes exhibit a strict 6T SLO. However, even in this case, μSteal improves instance count by 1.12× relative to dedicated instance pools. μSteal allows for a higher degree of sharing to accomodate transient load variations among different classes (i.e., when one class has few pending requests, the other may have many, which then utilize the otherwise idle cores). In this case, the FCFS policy also performs almost the same as μSteal as, if the two classes share the same latency SLO, FCFS reduces to EDF (all requests have the same "deadline"). Static core partitioning, however, still performs poorly in this scenario, due to lack of work conservation.

The results follow the same trend in both microservices. However, μSteal's improvements are noticably smaller for the image search microservice, since the image search service time (O(10μs)) is drastically shorter than the speech recognition microservice service time (O(10ms)) magnifying the impact of preemption.

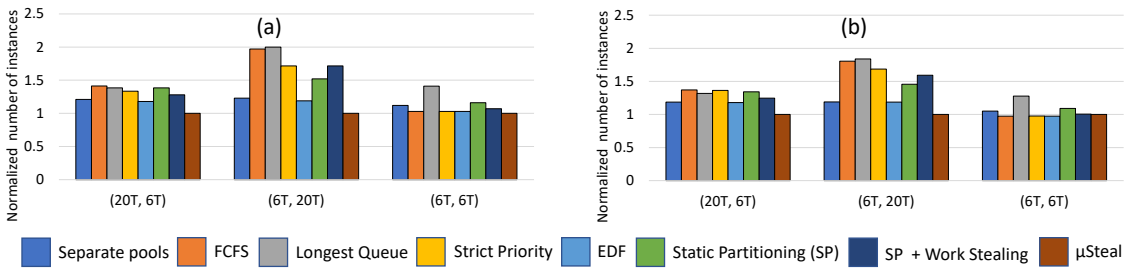


Figure 6.7: Normalized total number of instances for for (a) speech recognition and (b) image search microservices for deploying separate instances as well as sharing the instances across deployments with different scheduling policies. The arrival rates for both classes are asymmetric wherein the first request class accounts for 75% and the second request class accounts for 25% of the traffic. Latency SLOs are denoted in (20T, 6T) for the 99^{th} percentile tail latency target of the (first, second) request class.

6.6.2 Asymmetric traffic

Figure 6.7 reports the maximum required instance count for the (a) speech recognition and (b) image search microservices for two request classes when the first class accounts for 75% of the arrival traffic and the second accounts for 25%. We consider a case where both classes require equally strict 6T SLO despite asymmetric arrival rates and the two alternative cases where one class exhibits a strict 6T latency SLO and the other a relaxed 20T SLO. Whereas the results generally track the symmetric traffic cases, we make a few key observations.

First, μSteal's improvements over almost all other scheduling policies are much larger in the (20T, 6T) case, while improvements in the (6T, 20T) case are smaller. In the (6T, 20T) case, the class that accounts for the bulk of the traffic also requires a tighter SLO. Hence, most scheduling policies already perform well regardless of how they optimize. As an example, the strict prioritization policy exclusively favors the 6T SLO class, but since that class already accounts for 75% of the traffic, only a small fraction of requests are disadvantaged. Similarly, static partitioning allocates more cores to the strict SLO class, which also accounts for the bulk of the traffic.

On the other hand, for the (20T, 6T) case, most scheduling policies find their optimization objectives at odds with one another, since the class that accounts for the bulk of the requests

is the one with the more relaxed SLO—most scheduling policies perform poorly in this scenario, magnifying μSteal's improvements. As an example, strict prioritization results in numerous SLO violations of the relaxed SLO class, as it starves the bulk of the requests behind the strict-SLO minority. The strict priority policy achieves an instance count $1.71\times$ higher than that of μSteal. In contrast, the longest queue policy always optimizes for the class with the higher arrival rate, and hence, always picks the next request from the class with the relaxed 20T SLO—this policy results in the highest instance count—$2.1\times$ higher than μSteal—as it fully de-prioritizes strict-SLO requests. μSteal, however, significantly improves instance count relative to all alternatives by reserving more cores to the class with the strict 6T SLO (19 of 24 cores for speech recognition; 20 of 24 for image search). While the 6T class reserves most cores, these cores are largely idle and may be stolen by the 20T class. However, by reserving few cores to the 20T class, its requests do not starve behind those from the 6T class, as in the strict priority policy.

Finally, for the case where both classes exhibit a strict 6T SLO, the results are similar to the case with symmetric traffic, with the exception that the longest queue policy performs much worse here, achieving an instance count $1.41\times$ higher than μSteal. This policy strictly prioritizes the higher–arrival-rate class, which is harmful when both classes require the same latency SLO.

6.6.3 Load spikes

To demonstrate the effectiveness of our analytical approach for tuning the reservations in response to load spikes (see Section 6.4), we design an experiment in Figure 6.8, wherein two classes share 10T latency SLOs and equal arrival rates (symmetric traffic) for the speech recognition microservice. We suppose that the auto-scaler has over-provisioned the instance count by 50% in anticipation of load spikes. As such, if the total arrival rate rises by 50%, the system still has sufficient capacity to service all requests without SLO violation (i.e., the system is over-provisioned by $2\times$ relative to the initial load). We compare the behavior of

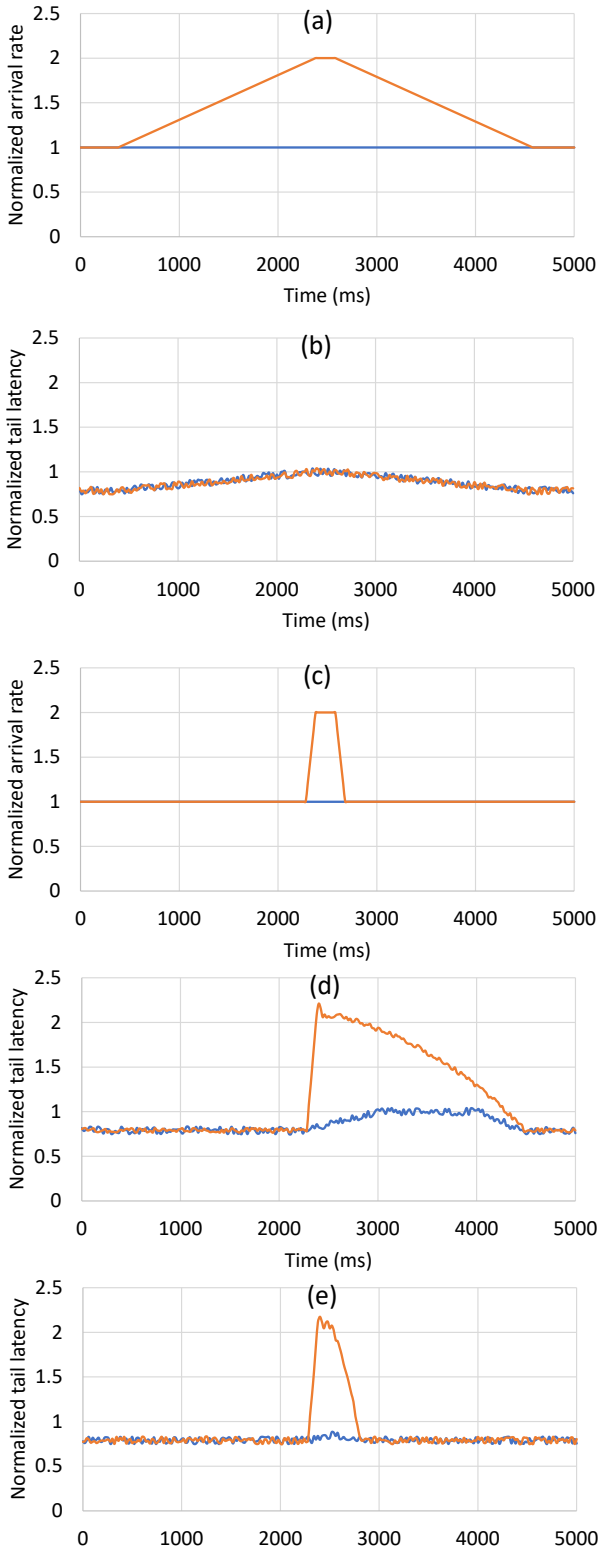


Figure 6.8: (a) Normalized arrival rate and (b) normalized tail latency, when B's load doubles gradually. (c) Normalized arrival rate, and (d)/(e) normalized tail latency, when B experiences a sudden $2\times$ load spike (d) without and (e) with the analytical reservation tuning mechanism of μSteal.

the μSteal with and without the analytical approach described in Section 6.4. Tail latencies reported in the figure are normalized to the SLO latency target (tail latencies above 1.0 indicate SLO violation).

Since both classes have the same latency SLO and the same arrival rate, each is allocated a 12 core reservation (half of the available cores). In all of our experiments, the arrival rate of class *A* remains fixed. First, in Figure6.8(a) we consider a case where the arrival rate of class *B* gradually increases by $2\times$ and returns to the original rate. As Figure 6.8(b) shows, neither class experiences significant SLO violations, as the feedback controller of Algorithm 2 has sufficient time to rebalance core reservations. This scenario does not even invoke the analytical reservation tuning method, as the changes in the arrival rate of class *B* does not exceed 20% in any epoch, and neither request class experiences any SLO violation; the system over-provisioning can absorb the load ramp.

Next, as shown in Figure 6.8(c), we consider a case where the arrival rate for class *B* doubles sharply for a short period of time. In Figure 6.8(d), we consider μSteal's behavior without the analytical reservation tuning approach (only the feedback controller). As shown in the figure, whereas the offered load spike subsides quickly, since the feedback controller fails to allocate sufficient processing capacity to *B* during the spike, the impact of the spike persists long after the load subsides, due to formation of long request queues during the spike. In contrast, Figure 6.8(e) considers a case where the analytical reservation tuning approach is invoked when the arrival rate spikes. As the figure shows, whereas *B* initially experiences a sharp latency increase, its tail latency rapidly recovers when the analytical approach re-tunes reservations, even before the load spike subsides.

Finally, we consider a case where class *A* experiences a $3\times$ load increase, which is beyond the reserve over-provisioned capacity of the system. As shown in Figure 6.9, while *B* experiences a higher tail latency than its SLO latency target (tail latency begins recovering when the auto-scaler upsizes the cluster), class *A* does not experience any persistent SLO violations, as it is allocated a dedicated service capacity through its own core reservations.

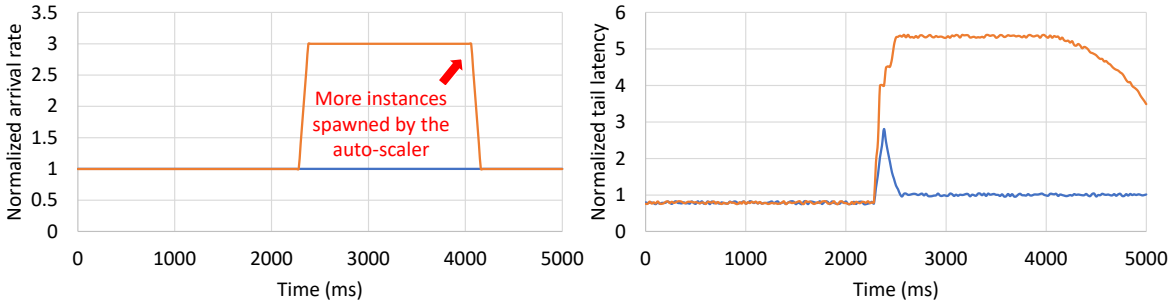


Figure 6.9: (a) Normalized arrival rate and (b) normalized tail latency (to the SLO latency target) for a scenario where *B*'s load is tripled.

Our result demonstrates the advantage of μSteal over FCFS scheduling even when all classes exhibit the same latency SLO—unlike Figures 6.6 and 6.7, where the total arrival rate is below the available service capacity, when one class experiences a sudden arrival rate spike, μSteal guarantees all other classes their reserved service capacity, guarding them from the spiking class.

6.7 Related Work

Scheduling for microservices. A large body of work has sought to reduce the tail latency by more efficiently scheduling requests within each instance of a microservice. Most of such works have focused on high disparity microservices which are prone to HoL blocking caused by rare, long tasks. Shunjuku [91] seeks to address this challenge via implementing a highly efficient preemption mechanism to enable *processor sharing* by eliminating the operating system threading overheads. RPCValet [36], Nebula [190], and Q-Zilla [140, 141] make the observation that shared request queues are very costly for μs-scale microservices despite being imperative for achieving minimal tail latency. They seek to enable shared queues through specialized hardware support. GrandSLAm [95] makes the observation that microservices are inherently different than classic services, due to their multi-stage nature. It proposes to use EDF scheduling to account for slowdowns and re-orderings in the previous microservice stages and execute requests at each stage based on their original

arrival order to the entire system, rather than their arrival order at a particular microservice. None of these systems consider mixed-criticality microservices with multiple SLOs for different request class, which μSteal seeks to address. μSteal is composable to all of these systems—whereas our implementation considers FCFS scheduling within the request queue of each class, these queues may instead implement processor sharing, EDF, etc.

Work stealing is a classic technique in high-performance computing applications, which seeks to maximize the execution throughput of parallel tasks. In such environments, tasks exhibit parent-child dependencies, which impose execution ordering. Due to such ordering constrains and localities across dependent tasks, it is advantageous in many cases to employ per-core—rather than shared—task queues, in addition to reducing the synchronization costs. A core only performs work stealing when its own task queue is empty, retaining the "work conserving" property of the system. MIT's Cilk [16] was the main framework that modernized this old idea and provided some provable properties around it. Since then, many frameworks have sought to optimize work stealing via providing better task queue implementations [26], alternative victim selection strategies [34], efficiently supporting reduction operations across tasks [113], as well as providing architectural support for work stealing in heterogeneous environments [13, 32, 194]. Nonetheless, server workloads usually employ shared—rather than per-core— queues, since there is no locality or dependency across requests, and the metric of interest is individual requests' response time, rather than throughput or latency of the entire job [139]. Tail-control [116] and ZygOS [162] are the only framework that leverage work stealing for server workloads. Both of them employ work-stealing to emulate shared queues at a lower cost. Tail control [116] targets parallelizable cloud workloads; when a request at the head of a local-queue is taking too long to be processed, another core steals fractions of it to accelerate its processing and reduce the wait time for the requests behind it. ZygOS [162] steals from the queue whose requests have already experienced a long wait time, perhaps due to HoL blocking. μSteal is different than these systems as it leverages work stealing across request classes's queues,

rather than per-core queues.

6.8 Conclusion

In this chapter we proposed μSteal as a scheduling framework for mixed-criticality microservices. *μSteal* leverages preemptive work and resource stealing to schedule the arriving requests to cores within an instance. μSteal provisions "core reservations" for each request class based on their latency requirements, but allows a class to steal cores from other classes if the cores would otherwise remain idle. We proposed a runtime feedback controller augmented by a queuing-theory based initialization approach to tune μSteal's reservation configuration. μSteal reduces the total number of instances required for a mixed-criticality microservices by $1.29\times$ as compared to deploying multiple instance pools, while ensuring all request classes meet their latency constraints.

CHAPTER VII

Conclusion

Hyperscale web services are moving towards loosely-coupled microservices that communicate via RPCs to improve programmability, reliability, and scalability of cloud software. Whereas microservice-based architectures have been adopted by serveral organizations and companies, they bring about many new challenges for computer system designers and architects. In this dissertation, we sought to address the some of such challenges by designing more efficient and performant hardware and runtime systems for microservices that particularly exhibit μs-scale service times.

We first addressed the problem of Killer Microseconds by introducing the Duplexity server architecture in Chapter II. Duplexity is a heterogeneous server architecture that employs aggressive multithreading to hide the latency of μs-scale I/O stalls and idle periods, without sacrificing the QoS of latency-sensitive microservices. Then, in chapters III-IV, we characterized different aspects of tail latency for microservices and sought to come up with effective solutions that minimize the tail latency at low cost. To this end, in chapter IV, we explored the Q-Zilla framework, which aims to tackle tail latency from a queuing perspective, by minimizing the probability of HoL blocking induced by rare, long tasks.

Finally, we explored the tail latency problem of microservices from a cluster, rather than server-level, perspective. We first introduced Parslo in Chapter V as a Gradient Descent-based framework for partial SLO allocation in virtualized cloud microservices, which

minimizes the total cost for the entire deployment of an end-to-end service given a microservice DAG. Then, in Chapter VI we proposed μSteal as a request scheduling framework for mixed-criticality microservices, which leverages preemptive work and resource stealing and seeks to maximize request throughput within an instance while ensuring all request classes meet their latency target.

Altogether, the systems presented in this dissertation (i.e., Duplexity, Q-Zilla, Parslo, and μSteal) synergistically improve the efficiency and performance of microservice-based cloud applications on modern hardware. Duplexity facilitates the execution of μs-scale microservices and improves their efficiency in the face of Killer Microseconds. Q-Zilla improves the tail latency of μs-scale microservices as the key metric that determines the user satisfaction in web services. Parslo find an optimal solution for allocating partial SLOs to microservices within and end-to-end service, which consequentially results in a minimized total deployment cost for the entire service. Finally, μSteal facilitates the execution of mixed-criticality microservices which may be shared across multiple end-to-end services but have to satisfy multiple SLOs and reduces the aggregated instance costs significantly, compared to a segregated deployment for such microservices.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644. ACM, 2017.

[2] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Symposium on Cloud Computing*. ACM, 2017.

[3] Dulcardo Arteaga and Ming Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage*. ACM, 2014.

[4] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino temporal data prefetcher. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[5] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo Spatial Data Prefetcher. In *International Symposium on High-Performance Computer Architecture*, 2019.

[6] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Evaluation of hardware data prefetchers on server processors. *ACM Computing Surveys (CSUR)*, 52(3):1–29, 2019.

[7] Nikhil Bansal and Mor Harchol-Balter. *Analysis of SRPT scheduling: Investigating unfairness*, volume 29. ACM, 2001.

[8] Mahmoud Barhamgi, Djamal Benslimane, and Brahim Medjahed. A query rewriting approach for web service composition. *IEEE Transactions on Services Computing*, 2010.

[9] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[11] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, number EPFL-CONF-201671. USENIX, 2014.

[13] Michael A Bender and Michael O Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems*, 35(3):289–304, 2002.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[15] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[16] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[18] Gérard Boudol. Fair cooperative multithreading. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 272–286, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[19] Onno Boxma and Bert Zwart. Tails in scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):13–20, 2007.

[20] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[21] Steve Byan, James Lentini, Anshul Madan, and Luis Pabon. Mercury: Host-side flash caching for the data center. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[22] Adrian Caulfield, Eric Chung, Andrew Putnam, et al. A cloud-scale acceleration architecture. In *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[23] Cavium. ThunderX ARM Processors. `https://cavium.com/product-thunderx-arm-processors.html`.

[24] Francisco J Cazorla, Peter MW Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Predictable performance in smt processors: Synergy between the os and smts. *IEEE Transactions on Computers*, 2006.

[25] Francisco J Cazorla, Alex Ramirez, Mateo Valero, Peter MW Knijnenburg, Rizos Sakellariou, and Enrique Fernández. Qos for high-performance smt processors in embedded systems. *Ieee Micro*, 2004.

[26] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, 2005.

[27] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[28] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and José F Martınez. Workload characterization of interactive cloud services on big and small server platforms. In *IEEE International Symposium on Workload Characterization*, 2017.

[29] Zeshan Chishti and TN Vijaykumar. Optimal power/performance pipeline depth for smt in scaled technologies. *IEEE Transactions on Computers*, 2008.

[30] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the killer microsecond. In *International Symposium on Microarchitecture 2018.*

[31] Chih-Hsun Chou, Laxmi N. Bhuyan, and Daniel Wong. μdpm: Dynamic power management for the microsecond era. In *IEEE International Symposium on High Performance Computer Architecture*, 2019.

[32] Kallia Chronaki, Alejandro Rico, Rosa M Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 329–338, 2015.

[33] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[34] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*, pages 57–66. IEEE, 2008.

[35] Mark E Crovella, Mor Harchol-Balter, and Cristina D Murta. Task assignment in a distributed system (extended abstract): improving performance by unbalancing load. In *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[36] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 35–48, New York, NY, USA, 2019. ACM.

[37] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[38] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*. ACM, 2013.

[39] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices*. ACM, 2014.

[40] Christina Delimitrou and Christos Kozyrakis. Amdahl's Law for Tail Latency. In *Communications of the ACM (CACM)*, August 2018.

[41] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Communications of the ACM*, 61(8):65–72, 2018.

[42] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association, Boston, MA*, 2019.

[43] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 431–442. ACM, 2014.

[44] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[45] Gautham K Dorai and Donald Yeung. Transparent threads: Resource sharing in smt processors for high single-thread performance. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 30–41. IEEE, 2002.

[46] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[47] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *European Conference on Computer Systems*. ACM, 2014.

[48] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ACM Sigplan Notices*, volume 45, pages 335–346. ACM, 2010.

[49] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, et al. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[50] Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael Abu-Ghazaleh. Corf: Coalescing operand register file for gpus. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019.

[51] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[52] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. *ACM Sigplan Notices*, 45(3):91–102, 2010.

[53] Facebook. Rocksdb. `https://rocksdb.org/`, 2018.

[54] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, 2013.

[55] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*. ACM, 2012.

[56] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004.

[57] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 2019.

[58] Yu Gan and Christina Delimitrou. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL), vol.17, iss. 2*, Jul-Dec 2018.

[59] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[60] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[61] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–26, 2012.

[62] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. Persistency for synchronization-free regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[63] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F Wenisch. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 337–350. ACM, 2019.

[64] Google. OpenImages: A public dataset for large-scale multi-label and multi-class image classification., howpublished = "`https://github.com/openimages/dataset`".

[65] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, 2017.

[66] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[67] Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to qos: case studies in cmp resource management. *ACM SIGARCH Computer Architecture News*, 35(1):21–30, 2007.

[68] Varun Gupta, Mor Harchol-Balter, JG Dai, and Bert Zwart. On the inapproximability of m/g/k: why two moments of job size distribution are not enough. *Queueing Systems*, 64(1):5–48, 2010.

[69] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 2017.

[70] Md E Haque, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM SIGPLAN Notices*. ACM, 2015.

[71] Md E Haque, Yuxiong He, Sameh Elnikety, Thu D Nguyen, Ricardo Bianchini, and Kathryn S McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *IEEE/ACM International Symposium on Microarchitecture*, 2017.

[72] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[73] Mor Harchol-Balter, Mark E Crovella, and Cristina D Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999.

[74] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems (TOCS)*, 21(2):207–233, 2003.

[75] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2015.

[76] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[77] Xin He and Yaacov Yesha. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation*, 1987.

[78] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[79] Sébastien Hily and André Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *International Symposium On High-Performance Computer Architecture*. IEEE, 1999.

[80] David A Holland, Elaine Lee Angelino, Gideon Wald, and Margo I Seltzer. Flash caching on the storage client. In *USENIX Annual Technical Conference*, 2013.

[81] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4):23–24, 2010.

[82] Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *IEEE International Symposium on High Performance Computer Architecture*, 2015.

[83] Intel. 3D Xpoint. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html`.

[84] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 186–197. ACM, 2007.

[85] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 25–36. ACM, 2007.

[86] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 314–325. ACM, 2010.

[87] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. Gpu register file virtualization. In *International Symposium on Microarchitecture*. ACM, 2015.

[88] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 155–168, New York, NY, USA, 2013. ACM.

[89] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 253–262. ACM, 2014.

[90] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *NSDI*, volume 14, pages 489–502, 2014.

[91] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 345–360, 2019.

[92] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 2015.

[93] Michael Kaminsky, Anuj Kalia Michael, and David G Andersen. Design guidelines for high performance rdma systems. In *USENIX Annual Technical Conference*, 2016.

[94] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE International Symposium on Computer Architecture*, 2015.

[95] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[96] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.

[97] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture*. ACM, 2015.

[98] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ACM SIGPLAN Notices*. ACM, 2014.

[99] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *IEEE International Symposium on Workload Characterization*, 2016.

[100] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. Regmutex: Inter-warp gpu register time-sharing. In *ACM/IEEE 45th International Symposium on Computer Architecture (ISCA)*, 2018.

[101] Khubaib Khubaib, M Aater Suleman, Milad Hashemi, Chris Wilkerson, Yale N Patt, et al. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *IEEE/ACM International Symposium on Microarchitecture*, 2012.

[102] Changkyu Kim, Simha Sethumadhavan, Madhu S Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W Keckler. Composable lightweight processors. In *IEEE/ACM International Symposium on Microarchitecture*, 2007.

[103] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds. In *HotStorage*, 2016.

[104] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[105] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing (ICAC)*, pages 51–60. IEEE, 2015.

[106] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Language-level persistency. In *ACM/IEEE International Symposium on Computer Architecture*, 2017.

[107] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE micro*, 2010.

[108] Łukasz Kruk, John Lehoczky, Kavita Ramanan, Steven Shreve, et al. Heavy traffic analysis for edf queues with reneging. *The Annals of Applied Probability*, 21(2):484–545, 2011.

[109] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

[110] Rakesh Kumar, Norman P Jouppi, and Dean M Tullsen. Conjoined-core chip multiprocessing. In *IEEE/ACM International Symposium on Microarchitecture*, 2004.

[111] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[112] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *International conference on World wide web*. ACM, 2010.

[113] I-Ting Angelina Lee, Aamir Shafi, and Charles E Leiserson. Memory-mapping support for reducer hyperobjects. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 287–297, 2012.

[114] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the workshop on Experimental computer science*. ACM, 2007.

[115] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM Symposium on Cloud Computing*, 2014.

[116] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I Lee, Chenyang Lu, Kathryn S McKinley, et al. Work stealing for interactive services to meet target latency. In *ACM SIGPLAN Notices*, volume 51, page 14. ACM, 2016.

[117] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM International Symposium on Microarchitecture*, 2009.

[118] A Likhtarov, Rajesh Nishtala, R McElroy, H Fugal, A Grynenko, and V Venkataramani. Introducing mcrouter: A memcached protocol router for scaling memcached deployments, 2014.

[119] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*. ACM, 2009.

[120] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 315–326. IEEE Computer Society, 2008.

[121] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM/IEEE International Symposium on Computer Architecture*, 2014.

[122] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[123] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. A performance study of big data on small nodes. *Proceedings of the VLDB Endowment*, 8(7):762–773, 2015.

[124] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. Scale-out processors. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2012.

[125] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *IEEE/ACM International Symposium on Microarchitecture*, 2012.

[126] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[127] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *IEEE International Symposium on High Performance Computer Architecture*, 2013.

[128] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[129] Simon J Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 131–140, 2011.

[130] Raman Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. Probabilistic shared cache management (prism). In *ACM SIGARCH computer architecture news*, volume 40, pages 428–439. IEEE Computer Society, 2012.

[131] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *International Symposium on Microarchitecture*. ACM, 2011.

[132] David Meisner, Brian T Gold, and Thomas F Wenisch. Powernap: eliminating server idle power. In *ACM Sigplan Notices*. ACM, 2009.

[133] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *International Symposium on Computer Architecture*. IEEE, 2011.

[134] David Meisner, Junjie Wu, and Thomas F Wenisch. Bighouse: A simulation infrastructure for data center systems. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE, 2012.

[135] Mellanox. ConnectX-3 VPI . `http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_VPI_Card.pdf`.

[136] Amirhossein Mirhosseini, Aditya Agrawal, and Josep Torrellas. Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery. *IEEE Computer Architecture Letters*, 16(2):153–157, 2017.

[137] Amirhossein Mirhosseini, Mohammad Sadrosadati, Behnaz Soltani, Hamid Sarbazi-Azad, and Thomas F Wenisch. Binochs: Bimodal network-on-chip for cpu-gpu heterogeneous systems. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2017.

[138] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Enhancing server efficiency in the face of killer microseconds. In *High Performance Computer Architecture (HPCA), 2019 IEEE 25th International Symposium on*. IEEE, 2019.

[139] Amirhossein Mirhosseini and Thomas F. Wenisch. The queueing-first approach for tail management of interactive services. In *IEEE MICRO*, 2019.

[140] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. Express-lane scheduling and multithreading to minimize the tail latency of microservices. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 194–199. IEEE, 2019.

[141] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 207–219. IEEE, 2020.

[142] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[143] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[144] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[145] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[146] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *International Symposium on High-Performance Computer Architecture*, 2003.

[147] Zwane Mwaikambo, Ashok Raj, Rusty Russell, Joel Schopp, and Srivatsa Vaddagiri. Linux kernel hotplug cpu support. In *Linux Symposium*, volume 2, 2004.

[148] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp scheduling. In *International Symposium on Microarchitecture*. ACM, 2011.

[149] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *International Symposium on Microarchitecture*, 2006.

[150] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[151] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. In *ACM SIGPLAN Notices*, volume 49, pages 3–18. ACM, 2014.

[152] Misja Nuyens and Bert Zwart. A large-deviations analysis of the gi/gi/1 srpt queue. *Queueing Systems*, 54(2):85–97, 2006.

[153] Oleander-Solutions. Oleander stemming library. `http://www.oleandersolutions.com/stemming/stemming.html`.

[154] Olumide Olusanya and Munira Hussain. Need for Speed: Comparing FDR and EDR InfiniBand. `http://en.community.dell.com/techcenter/high-performance-computing/b/general_hpc/archive/2016/02/02/need-for-speed-comparing-fdr-and-edr-infiniband-part-1`.

[155] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[156] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *ACM SIGPLAN Notices*, volume 53, pages 679–692. ACM, 2018.

[157] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *ACM SIGARCH Computer Architecture News*, 2014.

[158] Vinicius Petrucci, Michael A Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *IEEE International Symposium on High Performance Computer Architecture*, 2015.

[159] Phil Calcado. Building products at soundcloud —part i: Dealing with the monolith. [Online; accessed 27-Apr-2018].

[160] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[161] Martin F Porter. Snowball: A language for stemming algorithms, 2001.

[162] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, number EPFL-CONF-231395, 2017.

[163] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 805–825, 2020.

[164] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1–33, 2018.

[165] Qualcomm. Qualcomm Centriq 2400. `https://www.qualcomm.com/products/qualcomm-centriq-2400-processor`.

[166] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *IEEE/ACM International Symposium on Microarchitecture*, 2006.

[167] Steven E Raasch and Steven K Reinhardt. The impact of resource partitioning on smt processors. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 15–25. IEEE, 2003.

[168] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[169] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S McKinley. Exploiting processor heterogeneity in interactive services. 2013.

[170] Phil Rogers and AC Fellow. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[171] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[172] Satish Kumar Sadasivam, Brian W Thompto, Ron Kalla, and William J Starke. Ibm power9 processor architecture. *IEEE Micro*, 37(2):40–51, 2017.

[173] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018.

[174] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*. ACM, 2011.

[175] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075, 2019.

[176] Akbar Sharifi, Shekhar Srikantaiah, Asit K Mishra, Mahmut Kandemir, and Chita R Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM, 2011.

[177] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: a near-memory content-aware page-merging architecture. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 302–314. ACM, 2017.

[178] Faissal M Sleiman and Thomas F Wenisch. Efficiently scaling out-of-order cores for simultaneous multithreading. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 431–443. IEEE Press, 2016.

[179] Allan Snavely and Dean M Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *ACM SIGPLAN Notices*, 2000.

[180] Stephen Somogyi, Thomas F Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News*, 2009.

[181] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *ACM SIGARCH Computer Architecture News*, 2006.

[182] Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. Sharp control: controlled shared cache management in chip multiprocessors. In *International Symposium on Microarchitecture*. ACM, 2009.

[183] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526. ACM, 2019.

[184] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F. Wenisch. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2016.

[185] Akshitha Sriraman and Thomas F Wenisch. μsuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*.

[186] Akshitha Sriraman and Thomas F Wenisch. μSuite: A Benchmark Suite for Microservices. In *International Symposium on Workload Characterization*. IEEE, 2018.

[187] Akshitha Sriraman and Thomas F. Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[188] Staci D. Kramer. The biggest thing amazon got right: The platform. [Online; accessed 27-Apr-2018].

[189] Steven Ihde and Karan Parikh. From a monolith to microservices + rest: the evolution of linkedin's service architecture. [Online; accessed 27-Apr-2018].

[190] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The nebula rpc-optimized architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 199–212. IEEE, 2020.

[191] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. Mqsim: a framework for enabling realistic studies of modern multi-queue ssd devices. In *USENIX Conference on File and Storage Technologies*, 2018.

[192] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gomez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, et al. Enabling efficient rdma-based synchronous mirroring of persistent memory transactions. *arXiv preprint arXiv:1810.09360*, 2018.

[193] Tony Mauro. Adopting microservices at netflix: Lessons for architectural design. [Online; accessed 27-Apr-2018].

[194] Christopher Torng, Moyang Wang, and Christopher Batten. Asymmetry-aware work-stealing runtimes. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 40–52. IEEE, 2016.

[195] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proceedings of the 2007 workshop on Experimental computer science*, page 4. ACM, 2007.

[196] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ACM SIGARCH Computer Architecture News*. ACM, 1996.

[197] Eric Tune, Rakesh Kumar, Dean M Tullsen, and Brad Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Microarchitecture, 2004. 37th International Symposium on*. IEEE, 2004.

[198] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1–39, 2008.

[199] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[200] Augusto Vega, Alper Buyuktosunoglu, and Pradip Bose. Smt-centric power-aware thread placement in chip multiprocessors. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 2013.

[201] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[202] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 31–40. IEEE, 2013.

[203] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 207–217. IEEE, 2017.

[204] Yasuko Watanabe, John D Davis, and David A Wood. Widget: Wisconsin decoupled grid execution tiles. In *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[205] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. *ACM SIGARCH Computer Architecture News*, 2005.

[206] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[207] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[208] Wikipedia-Redux. `https://reagle.org/joseph/blog/social/wikipedia/10k-redux.html`.

[209] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International*

Symposium on High Performance Computer Architecture (HPCA), pages 331–344. IEEE, 2019.

[210] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *IEEE International Symposium on High Performance Computer Architecture*, 2015.

[211] Yuejian Xie and Gabriel H Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009.

[212] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 329–341, 2013.

[213] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 607–618. ACM, 2013.

[214] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Computer performance microscopy with shim. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 170–184. IEEE, 2015.

[215] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX Annual Technical Conference*, 2016.

[216] Yoni Goldberg. Scaling gilt: from monolithic ruby application to distributed scala micro-services architecture. [Online; accessed 27-Apr-2018].

[217] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 2016.

[218] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *OSDI*, 2014.

[219] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.

[220] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 406–418. IEEE Computer Society, 2014.

[221] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 456–468. IEEE, 2016.

[222] Yanqi Zhou and David Wentzlaff. The sharing architecture: sub-core configurability for iaas clouds. *ACM SIGARCH Computer Architecture News*, 2014.

[223] Yanqi Zhou and David Wentzlaff. Mitts: memory inter-arrival time traffic shaping. In *ACM SIGARCH Computer Architecture News*. IEEE, 2016.

[224] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016.